

## Tutorial-00 (versión Python): Instalación del software y primeros pasos.

- Este documento pdf lleva adjuntos algunos de los ficheros de datos necesarios. Y está pensado para trabajar con él directamente en tu ordenador. Al usarlo en la pantalla, si es necesario, puedes aumentar alguna de las figuras para ver los detalles. Antes de imprimirlo, piensa si es necesario. Los árboles y nosotros te lo agradeceremos.
- Fecha: 14 de mayo de 2016. Si este fichero tiene más de un año, puede resultar obsoleto. Busca si existe una versión más reciente.

## Índice

<b>1. Lectores de documentos en formato pdf.</b>	<b>2</b>
<b>2. Navegador de Internet.</b>	<b>4</b>
<b>3. Instalación de la hoja de cálculo Calc.</b>	<b>5</b>
<b>4. Editores de texto.</b>	<b>9</b>
<b>5. Ficheros csv con Calc.</b>	<b>10</b>
<b>6. Instalación de Anaconda Python. Python2 vs Python 3.</b>	<b>16</b>
<b>7. Instalación de GeoGebra.</b>	<b>23</b>
<b>8. Siguiente paso. ¿Dónde vamos ahora?</b>	<b>28</b>

En este primer tutorial vamos a limitarnos a preparar las herramientas que necesitamos para el trabajo del curso, que comenzará realmente a partir del Tutorial-01. Instalaremos casi todos los programas que vamos a utilizar, y repasaremos algunas cuestiones de índole práctica que conviene discutir al principio, antes de que nos causen problemas más adelante. Es posible que, dependiendo de tu experiencia previa con ordenadores, todas o algunas de esas cuestiones te resulten muy fáciles. Las incluimos aquí porque, en nuestra experiencia, inicialmente causan problemas a muchos estudiantes del curso.

Vamos a describir paso a paso, y con capturas de pantalla, la instalación de varios programas. Te aconsejamos que vayas *un paso por delante* en la lectura, en lugar de ir ejecutando las cosas a la vez que lees las instrucciones, para evitarte sorpresas o errores.

## Estructura de directorios para los ficheros del curso.

A lo largo del curso vamos a manejar distintos tipos de ficheros. Para empezar, los propios ficheros pdf del libro y de estos tutoriales. Pero también usaremos ficheros de datos, ficheros con código Python, etc. Y además algunos de esos ficheros dependen, para su correcto funcionamiento, de que los ficheros del curso se encuentren *en el lugar esperado*. Podríamos haber optado por pedirte que crearas una carpeta<sup>1</sup> en tu ordenador en la que guardar, entremezclados, todos los ficheros del curso. Pero la experiencia ha demostrado que esa es una mala forma de trabajar, que resulta poco eficiente. Por esa razón vamos a pedirte que empieces el trabajo con una pequeña tarea de organización.

1. Empieza por elegir o crear un directorio en tu ordenador, al que de ahora en adelante nos referiremos como el **directorio de trabajo**. No es estrictamente necesario, pero lo más sencillo es que sea una carpeta nueva y puedes darle el nombre que quieras (te sugerimos PostData

<sup>1</sup>En lo que sigue usaremos indistintamente las palabras **directorio** (inglés, *directory*) o **carpeta** (inglés, *folder*).

como nombre). Este directorio puede servirte, por ejemplo, para almacenar los ficheros pdf del libro y de estos tutoriales. Pero el papel más importante de esa carpeta será servir como contenedor de las carpetas que indicamos a continuación.

2. Ahora vamos a crear una carpeta llamada *datos* (exactamente así, en minúsculas) que **debe** estar situada dentro de tu *directorio de trabajo*.
3. De la misma forma, vamos a crear otra subcarpeta de tu *directorio de trabajo* llamada *codigo* (exactamente así, en minúsculas y sin acento).

En cada momento, cuando sea necesario a lo largo de estos tutoriales, te iremos indicando en cuál de estas carpetas debes colocar cada fichero del curso para que todo funcione correctamente.

## 1. Lectores de documentos en formato pdf.

Si estás leyendo este tutorial en la versión pdf, en la pantalla de tu ordenador (que es la forma que recomendamos), eso significa que ya tienes instalado un lector de ficheros pdf. En sistemas Windows, el más extendido de estos programas es el Adobe Reader. Si usas este programa, te recomendamos que verifiques que tienes la última versión (esto es una buena idea, hablando en general, para cualquier programa que uses). Puedes usar el enlace:

<http://get.adobe.com/es/reader/>

De hecho, las últimas versiones para Windows, desde la 10.1, incluyen la opción de autoactualización del programa. Si tu versión es más antigua, ¡actualízala ahora mismo!

La mayoría de los ficheros pdf del curso, como este, incluirán:

- Enlaces a páginas web.
- Ficheros adjuntos, de los tipos que vamos a usar en el curso: hojas de cálculo, ficheros csv con datos, ficheros con código R, ficheros de GeoGebra, etc. Es decir, el fichero pdf contiene esos documentos, y no es preciso descargarlos de Internet por separado.

El lector pdf que utilices puede influir de forma importante en la facilidad de uso de esos documentos adjuntos. Nuestro favorito, para el sistema Windows, es el programa gratuito y de código abierto llamado SumatraPDF, que puedes descargar desde este enlace:

<http://blog.kowalczyk.info/software/sumatrapdf/free-pdf-reader-es.html>

Una primera advertencia:

### Enlaces y descarga de programas.

Suponemos que el usuario de este curso es consciente de que tiene que velar por su propia seguridad. Usa sólo los enlaces que te proporcionamos para descargar los programas. Hay una cantidad ingente de páginas de descarga de programas en la red. Y como nos explicó nuestra madre sobre los extraños que te encuentras en la calle, no todos son bien intencionados... En caso de duda, antes de instalar nada, consulta con alguien de confianza. Mantén tu ordenador actualizado, usa un buen antivirus, etc.

SumatraPDF no es el lector de PDF más conocido, ni el que más posibilidades ofrece, pero es muy rápido y ligero (en términos de consumo de recursos, si tu ordenador no es muy potente). Y se lleva bastante bien con los enlaces y ficheros adjuntos que incluiremos. Un simple click con el botón izquierdo del ratón basta para:

- Abrir un enlace en el navegador.
- Abrir un cuadro de diálogo para guardar un fichero adjunto.

La instalación de SumatraPDF es extremadamente sencilla, así que no creemos necesario entrar en detalles.

### 1.0.1. Programas predeterminados y usuarios “de gatillo fácil”.

Queremos extendernos un poco más sobre este último punto. Nuestra experiencia indica que los usuarios, con demasiada frecuencia, somos *de gatillo fácil* con el ratón: nos hemos acostumbrado a hacer click, o doble click, con el botón izquierdo del ratón con demasiada facilidad. En muchos casos, eso conduce a que el sistema trate de abrir el fichero adjunto con el *programa predeterminado* del sistema operativo para ese tipo de archivos. En la mayoría de los sistemas, el nombre de los archivos se compone de dos partes, separadas por un punto, como en este ejemplo:

ficheroDatos.csv  
identificador extensión

La extensión, como seguramente sabes, es el código (muchas veces, pero no siempre) de tres letras que sigue al punto, y que nos permite identificar el tipo de fichero. Pero, además, el tipo de fichero (la extensión), sirve para determinar cuál es el programa que se utiliza por defecto cuando abrimos el fichero, por ejemplo haciendo doble click sobre él con el ratón. De esa forma, al hacer doble clic sobre un fichero de tipo **csv**, nos podemos encontrar con la sorpresa de que el sistema intenta abrirlo con la hoja de cálculo Microsoft Excel, si está instalada, porque el sistema tiene una lista de asociaciones de ficheros con programas que dice cosas como

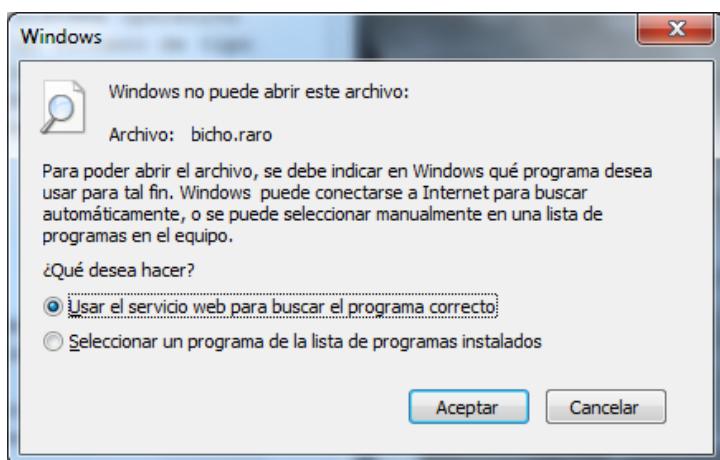
“los ficheros **csv** (cuya extensión es **.csv**) se abren con *Excel*”.

Es decir, *Excel* es el *programa predeterminado* (o programa por defecto) del sistema para ficheros de tipo **csv**. En general, este comportamiento del sistema es cómodo y nos ahorra tiempo. Pero, en ocasiones, eso no es lo que queremos que suceda. En particular, en este curso, nunca vamos a usar Excel para abrir ficheros **csv** (entre otras cosas, no asumimos que el usuario tenga Excel instalado). Para evitar eso, sigue este consejo:

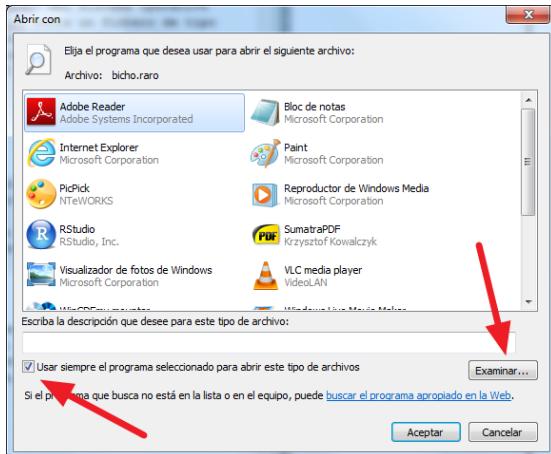
#### Procedimiento para abrir ficheros adjuntos:

1. Acostúmbrate a usar el botón derecho del ratón y busca opciones como **Guardar archivo...** Si eso no funciona, entonces y sólo entonces, prueba a hacer click o doble click con el botón izquierdo.
2. Una vez que hayas guardado el fichero en una carpeta de tu ordenador, tienes que abrirlo. Y aquí, de nuevo, hay que acostumbrarse a usar el botón derecho, y buscar opciones como **Abrir con...**

Si no hay disponible una opción como **Abrir con...**, se debe a que tu ordenador no tiene asignado un programa por defecto con el que abrir ese tipo de ficheros. Por ejemplo, en mi *Escritorio* de Windows (versión 7) tengo un fichero que se llama **bicho.raro**. Usando el botón derecho no aparece la opción **abrir con**, solo la opción **Abrir**. Usando esta opción (o si, directamente, hago doble click sobre el fichero) aparece esta ventana:



Lo mejor, en la inmensa mayor parte de los casos, es seleccionar la opción **Seleccionar un programa de la lista de programas instalados** y pulsar en **Aceptar**. En la ventana de diálogo que aparece a continuación, puedes seleccionar el programa que deseas utilizar. Pero tienes que prestar especial atención a los dos elementos que hemos indicado con flechas rojas en la figura.



La casilla **Usar siempre el programa...** es especialmente importante, porque puede cambiar el comportamiento de tu equipo, y tal vez no deseas ese cambio. ¡Así que ve con cuidado! Si esa casilla está marcada, y seleccionas el programa *A* (el que quieras) para abrir un fichero de tipo *B*, Windows modificará la lista a la que aludíamos antes, y escribirá en ella una línea

*“los ficheros de tipo B se abren por defecto con el programa A”.*

Si no quieres que pase eso, debes desmarcar esta casilla. Por lo demás, si el programa que deseas utilizar aparece en la ventana de la parte superior del cuadro de diálogo, basta con seleccionarlo y pulsar **Aceptar**. Cuando no es así, hay que usar el botón **Examinar...**, para localizar el programa que queremos usar. Esta parte puede ser más o menos fácil, dependiendo del programa que se trate, y de tu versión de Windows. Si tienes problemas para encontrar el programa, busca en Internet, o pide ayuda a alguien que sepa más que tú. En general ese consejo sirve no sólo para este paso, sino para cualquiera de los siguientes. Siempre conviene tener un ninja informático a mano.



## 2. Navegador de Internet.

Para muchas de las tareas asociadas a este curso, la elección de uno u otro navegador de Internet es irrelevante, siempre que se trate de versiones recientes. Pero para algunos temas concretos del curso es recomendable que utilices el navegador Firefox, que puedes descargar desde este enlace:

<http://www.mozilla.org/es-ES/firefox/new/>

Hay versiones disponibles para Windows, Mac y Linux. La razón por la que te recomendamos Firefox es porque este navegador permite visualizar correctamente las fórmulas matemáticas, mientras que otros navegadores nos han causado más problemas al hacer esto. En cualquier caso, aparecen nuevas versiones de los navegadores muy a menudo. Y esas nuevas versiones pueden corregir algunos de esos problemas (desdichadamente, hemos tenido también experiencia con el proceso contrario, en el que una nueva versión estropeaba algo que ya estaba funcionando). Así que si quieras comprobar si tu navegador funciona correctamente puedes visitar esta página web:

<https://www.tuhh.de/MathJax/test/sample.html>

Espera unos segundos y asegúrate de que en tu navegador aparecen las fórmulas matemáticas como en esta figura:

## Sample MathJax Equations

The Lorenz Equations

$$\begin{aligned}\dot{x} &= \sigma(y - x) \\ \dot{y} &= \rho x - y - zx \\ \dot{z} &= -\beta z + xy\end{aligned}$$

The Cauchy-Schwarz Inequality

$$\left(\sum_{k=1}^n a_k b_k\right)^2 \leq \left(\sum_{k=1}^n a_k^2\right) \left(\sum_{k=1}^n b_k^2\right)$$

A Cross Product Formula

$$\mathbf{v}_1 \times \mathbf{v}_2 = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \frac{\partial X}{\partial u} & \frac{\partial Y}{\partial u} & 0 \\ \frac{\partial X}{\partial v} & \frac{\partial Y}{\partial v} & 0 \end{vmatrix}$$

The probability of getting  $k$  heads when flipping  $n$  coins is:

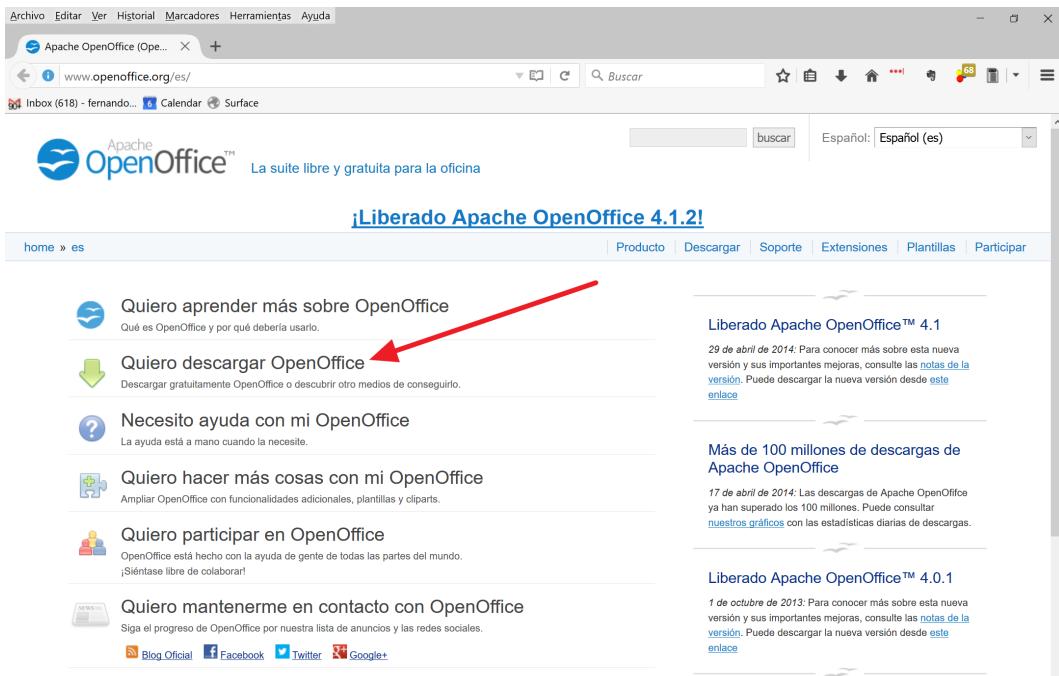
$$P(E) = \binom{n}{k} p^k (1-p)^{n-k}$$

### 3. Instalación de la hoja de cálculo Calc.

El siguiente paso es instalar, si no dispones ya de ella, la suite ofimática OpenOffice, que incluye la hoja de cálculo Calc<sup>2</sup>, que vamos a utilizar, especialmente al principio del curso. Para ello dirígete a

<http://www.openoffice.org/es/>

y usa el enlace *Quiero descargar OpenOffice*:

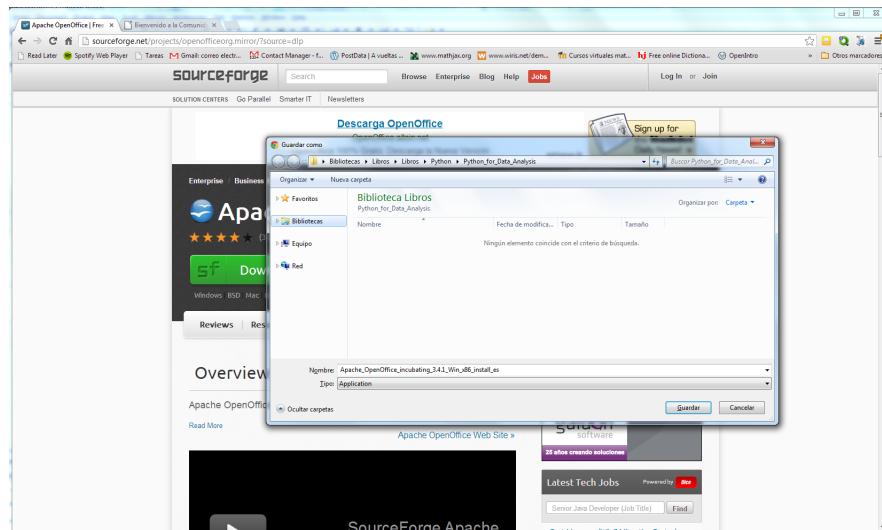


Usando ese enlace, se abrirá la ventana que aparece en la siguiente figura, en la que debes hacer click en el enlace indicado por la flecha. ¡Asegurate de que seleccionas tu sistema operativo y el idioma español! El número de versión habrá cambiado, desde luego. En la Figura aparece la versión 4.1.2, pero en el momento en que tú la descargas, posiblemente habrá avanzado:

<sup>2</sup>Si tienes instalado o prefieres instalar LibreOffice, no encontrarás apenas diferencia con OpenOffice, en lo que se refiere a este curso.

The screenshot shows the Apache OpenOffice download page. At the top, there's a navigation bar with links like Archivo, Editar, Ver, Historial, Marcadores, Herramientas, Ayuda, and a search bar. Below the header, the Apache OpenOffice logo and slogan 'La suite libre y gratuita para la oficina' are displayed. The main content area has a green header 'Descargar Apache OpenOffice' with a note about it being hosted on SourceForge.net. It asks to select an operating system (Windows EXE) and language (Español (Spanish)). Two buttons are shown: 'Descargar la instalación completa' (highlighted with a red arrow) and 'Descargar paquete de idioma'. To the right, there's a sidebar with sections like 'Información de la versión', 'Documentación', and 'Recursos adicionales'. At the bottom, there's a section for extensions and dictionaries.

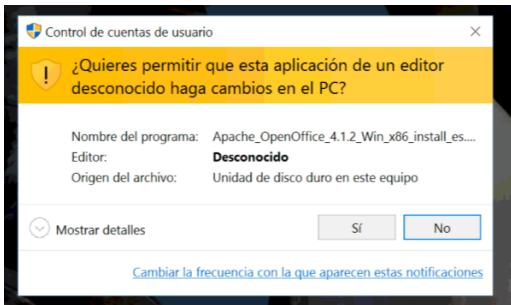
Con eso llegamos a la página de descarga (alojada en el dominio `sourceforge.net` a fecha de hoy) y en pocos segundos, según la configuración del navegador, se descargará el archivo automáticamente, o debe abrirse un cuadro de diálogo para guardar el fichero en alguna carpeta de tu ordenador (por ejemplo, *Descargas* en máquinas Windows). Lo más importante en este paso es que sepas en qué carpeta se guarda ese fichero, pero eso depende de tu configuración particular.



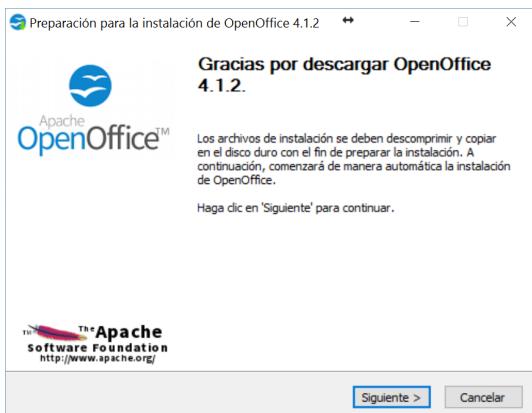
El fichero que has descargado se llamará (en Windows) algo parecido a:

`Apache_OpenOffice_incubating_4.1.2_Winx_86_install_es.exe`

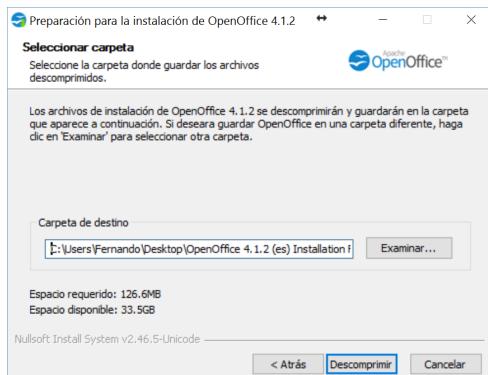
(aunque puede que no veas la extensión `.exe` en el Explorador de Windows). Ahora tienes que abrir ese fichero, para instalar el programa (usa el botón derecho otra vez). Para este paso, es necesario disponer de permisos de administración en el ordenador (de nuevo, si te pierdes, busca al ninja...). En las últimas versiones de Windows, al hacer esto la pantalla se oscurece y aparece un cuadro de diálogo que pregunta *¿Desea permitir que este programa realice cambios...?*. Debes pulsar en **Sí** para continuar la instalación (insistimos, en las próximas figuras el número de versión que aparecerá será otro, pero el proceso será esencialmente el mismo).



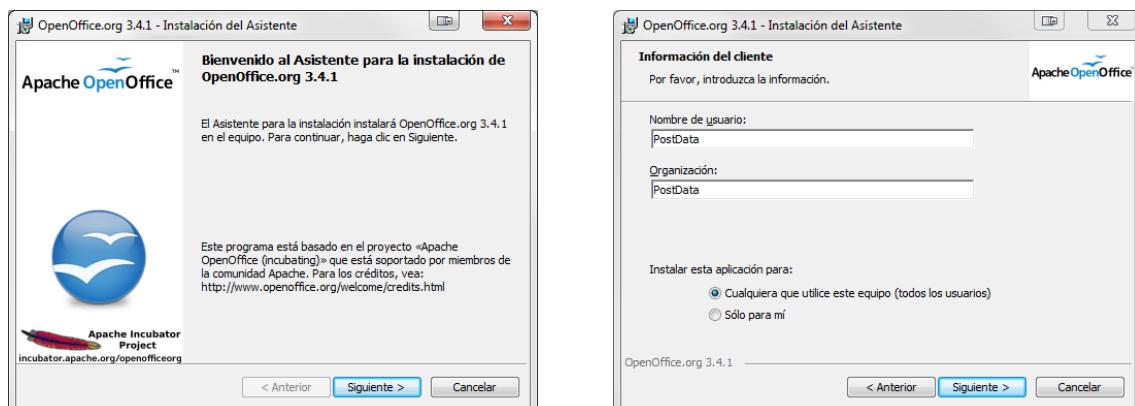
Empieza la instalación:

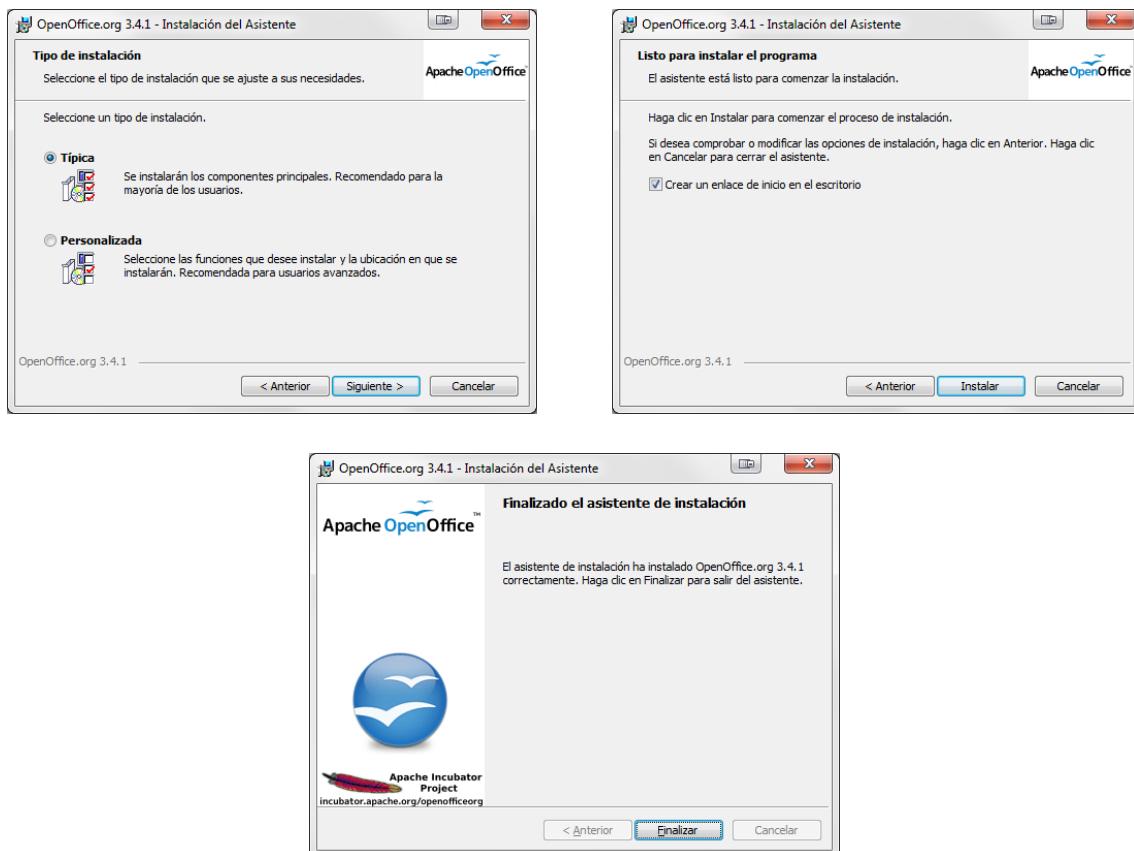


La siguiente ventana te preguntará dónde quieres guardar una carpeta con los ficheros *temporales* de instalación. Es importante, de nuevo, que recuerdes donde los guardas. Cuando termine la instalación puedes borrar esa carpeta, sólo es necesaria durante la instalación.

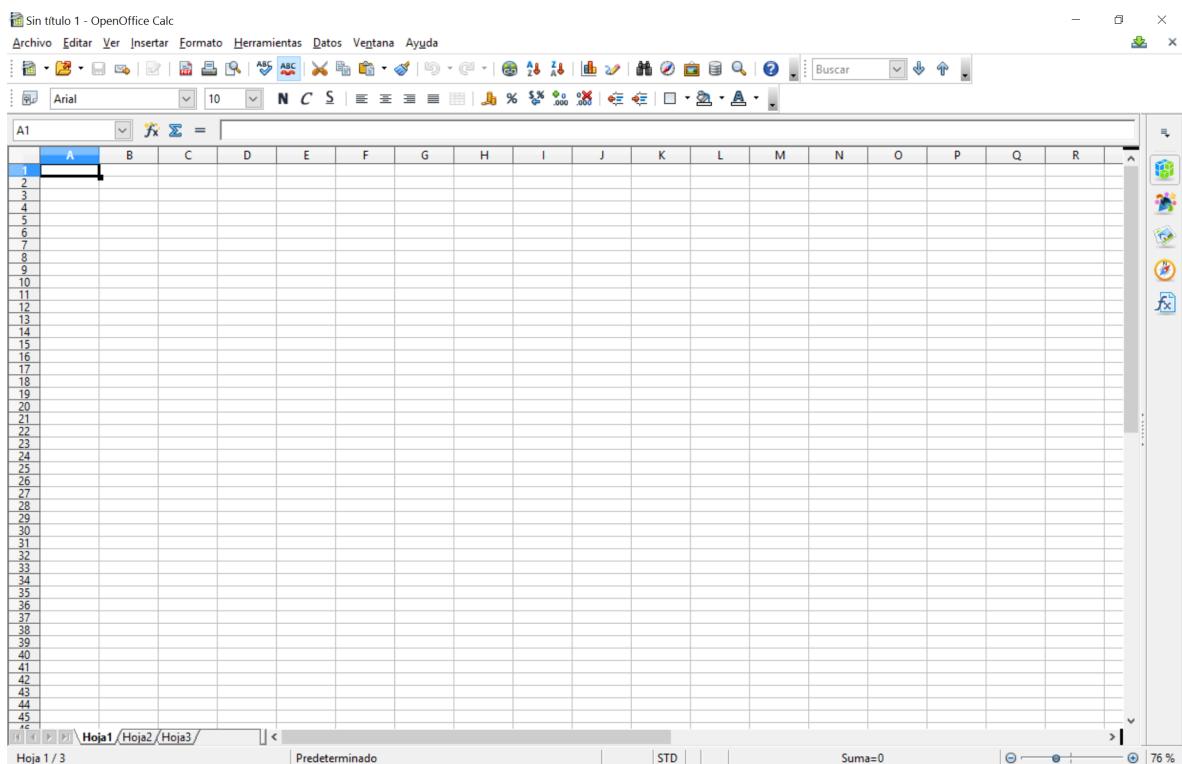


A continuación el programa va pasando por pantallas similares a estas (son de una versión anterior), en las que puedes, sin riesgos, aceptar todas las opciones por defecto (en la segunda, si escribes tu nombre de usuario, se incorporará a todos los documentos que crees con OpenOffice; puedes omitir esa información sin problemas):





Al llegar a esta última ventana pulsa en **Finalizar**, y la instalación habrá acabado. Ahora, para comprobar que todo ha ido bien, deberías buscar en la lista de programas del menú Inicio (de nuevo hablamos de Windows, aunque en otras plataformas es similar) el grupo de programas *OpenOffice*, y abrir el que se llama *OpenOffice.org Calc*. Tras una ventana de presentación y unos momentos, te encontrarás con esta pantalla (puedes verlo más o menos grande, según tu resolución de pantalla):

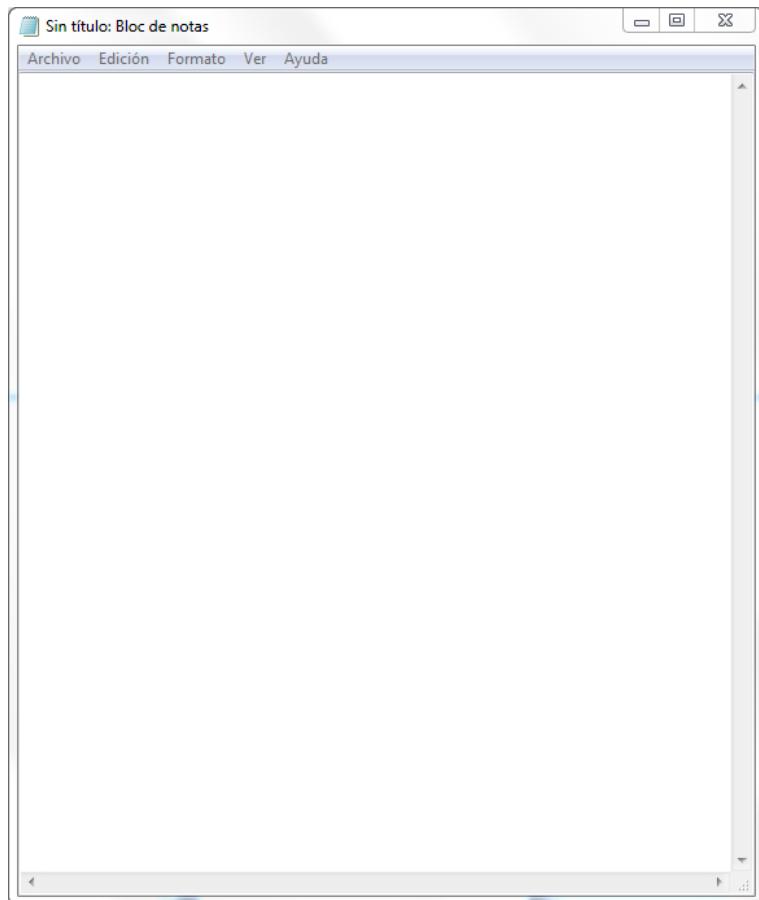


que indica que todo ha ido bien. Ya estamos listos para pasar al segundo apartado de este tutorial.

## 4. Editores de texto.

Nuestro objetivo, en esta sección, es localizar un *editor de texto*, como el *Bloc de Notas* en Windows, y aprender a usarlo para abrir ficheros **csv** (no te preocunes, enseguida aprenderemos qué son estos ficheros). En segundo lugar, vamos a aprender a abrir ficheros de tipo **csv** con Calc, eligiendo las opciones correctas en el menú de importación.

Empecemos por los editores de texto. En Windows, como ya hemos dicho, dispones del *Bloc de Notas*. Si no lo localizas fácilmente, pulsa simultáneamente las teclas **Windows** y **R**, y en el cuadro de diálogo que se abrirá escribe **NotePad**. Tras pulsar en Aceptar se abrirá el *Bloc de Notas* que, inicialmente tiene este aspecto:

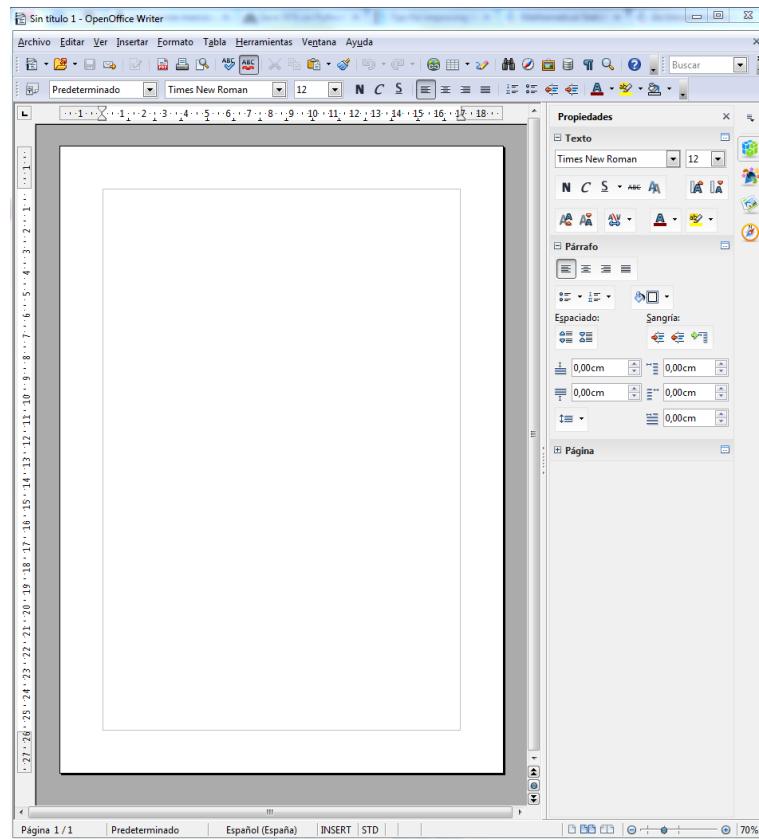


En Mac OS puedes usar el programa gratuito **TextWrangler**, que se descarga desde el enlace:

<http://www.barebones.com/products/textwrangler/>

TextEdit viene instalado en los Macs, pero no es exactamente un *editor de texto*, en el sentido que aquí le damos a esa expresión (ver más abajo) y seguramente te causaría algunos problemas más adelante. Y si eres usuario de Linux, a buen seguro ya conocerás algún editor de texto: (**kate**, **gedit**, **leafpad**, elige tu favorito).

Es importante que entiendas la diferencia entre los *procesadores de texto* y los *editores de texto*. Un procesador de texto es un programa diseñado para la elaboración de textos, con un enfoque esencialmente visual. El texto se puede formatear, cambiando el tipo y tamaño de letra, la tipografía (negrita, cursiva, subrayado), insertando imágenes, etc. El ejemplo más conocido es el programa *Word* de *Microsoft*. Al instalar *OpenOffice* en la sección anterior hemos instalado otro procesador de texto, llamado *Writer*. En la siguiente figura puedes ver el aspecto inicial de *Writer*, al abrir el programa, y compararlo con el del *Bloc de Notas*, que hemos visto antes.



El contraste entre el procesador de texto, lleno de herramientas de formato, y el aspecto casi vacío del editor de texto, debería ser evidente. Naturalmente, hay editores de texto más sofisticados que el *Bloc de Notas* (por ejemplo, en Windows, [Notepad++](#)), pero lo más importante es que comprendas que los procesadores de texto *no son adecuados* para el trabajo con los ficheros que vamos a usar en este curso, que son ficheros de *texto plano*. Los ficheros de texto plano más conocidos son los de extensión `txt`, pero hay muchos otros tipos. Por ejemplo, los ficheros de datos de tipo `csv` que vamos a ver a continuación. Pero también son ficheros de texto plano los ficheros de *código fuente* (en inglés, *source code*) de la mayoría de lenguajes de programación. Nosotros, en este curso, vamos a usar ficheros de código para el programa R, que serán ficheros de texto plano, con la extensión `.R`.

## 5. Ficheros csv con Calc.

Un fichero `csv` es un fichero de texto plano que contiene una tabla de datos. El nombre proviene del inglés, *comma separated values* (valores separados por comas, aunque ya veremos que no hay que tomarse el nombre al pie de la letra). Para empezar, vamos a trabajar con el fichero (que también usaremos en el Tutorial-01)

[Tut01-PracticaConCalc.csv](#).

Te aconsejamos que *guardes* el fichero, en lugar de *abrirlo* directamente. Recuerda lo que hemos visto en la Sección 1: el fichero de datos va *adjunto* a este documento pdf y, para guardar los datos en tu ordenador, debes hacer click (aquí mismo, en el documento pdf) sobre el nombre del fichero. ¿Click derecho o izquierdo? Depende del lector de pdfs que estés usando. ¡Recuerda que en muchos casos es mejor usar primero el botón derecho del ratón! El lugar adecuado para guardar este fichero es la subcarpeta **datos** de tu *Directorio de trabajo* (recuerda la estructura de carpetas que hemos creado al principio del tutorial), porque esa es la carpeta donde guardaremos todos los ficheros `csv` que vamos a usar en el curso. Si no estás muy seguro de haberlo hecho bien, este es otro paso en el que es posible que te pierdas. Si eso sucede, será un buen momento para acudir a nuestro amigo. Y, en cualquier caso, recuerda que también puedes descargar todos los ficheros adjuntos del curso desde la página web del curso, a la que se llega mediante este enlace:

<http://www.postdata-statistics.com/>.



Los ficheros **csv** se usan para guardar datos de una forma sencilla, en ficheros de texto, facilitando así el intercambio de datos entre programas. El fichero **Tut01-PracticaConCalc.csv** es un ejemplo típico: contiene una tabla de datos con tres columnas, y 1300 filas. Es una buena idea que empieces por abrirlo con un editor de texto (el *Bloc de Notas* en Windows, o similar) para hacerte una idea del aspecto que tienen los datos, pero no hagas ningún cambio en el fichero. En la siguiente figura puedes ver el aspecto de ese fichero cuando se abre con el *Bloc de Notas* de Windows.

```

var1 var2 var3
A 54,717 4
E 52,676 8
A 7,278 4
E 1,253 4
C 24,436 5
B 82,398 5
F 94,411 3
E 17,865 6
D 27,52 6
F 14,274 2
A 61,88 4
A 22,722 4
C 95,965 3
B 39,324 3
D 7,697 3
C 90,413 2
C 27,803 6
E 3,667 4
B 82,971 5
D 12,873 2
C 24,736 5
F 90,227 6
E 57,626 5
D 43,317 2
D 48,753 6
E 85,698 4
C 67,137 5
C 40,335 3
C 5,114 4
F 66,487 4
C 64,502 4
F 68,473 10
C 93,551 6

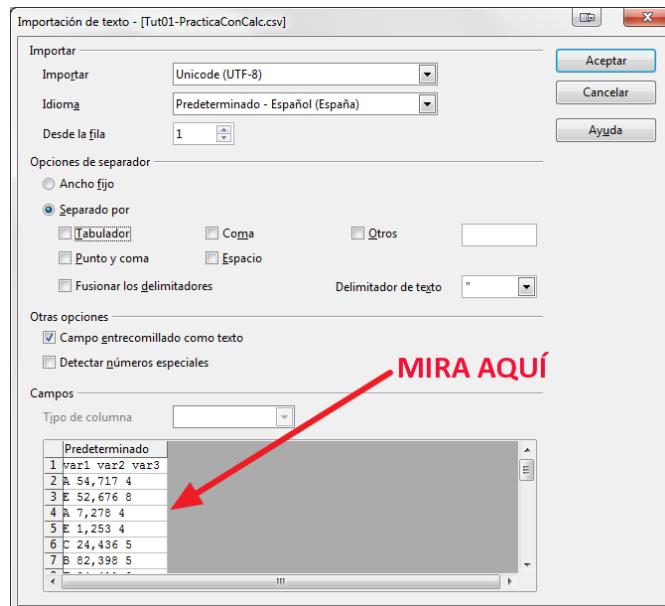
```

En este fichero en particular, hay guardada una tabla de tres columnas. Cada fila de la tabla se corresponde con una línea del fichero, y los elementos de las distintas columnas están separados por espacios. La primera línea es especial, porque contiene los nombres de las variables que corresponden a cada columna, y que son **var1**, **var2** y **var3**. Usando el editor de texto podemos ver los datos que contiene el fichero, e incluso hacer algunas modificaciones muy interesantes. Por ejemplo, podemos reemplazar todas las comas por puntos o viceversa. Pero el procesador de texto no sirve para analizar los datos desde el punto de vista estadístico. Para eso necesitamos herramientas más especializadas, como la hoja de cálculo, que vamos a ver a continuación; o programas específicos de Estadística, como R, que veremos en próximos tutoriales.

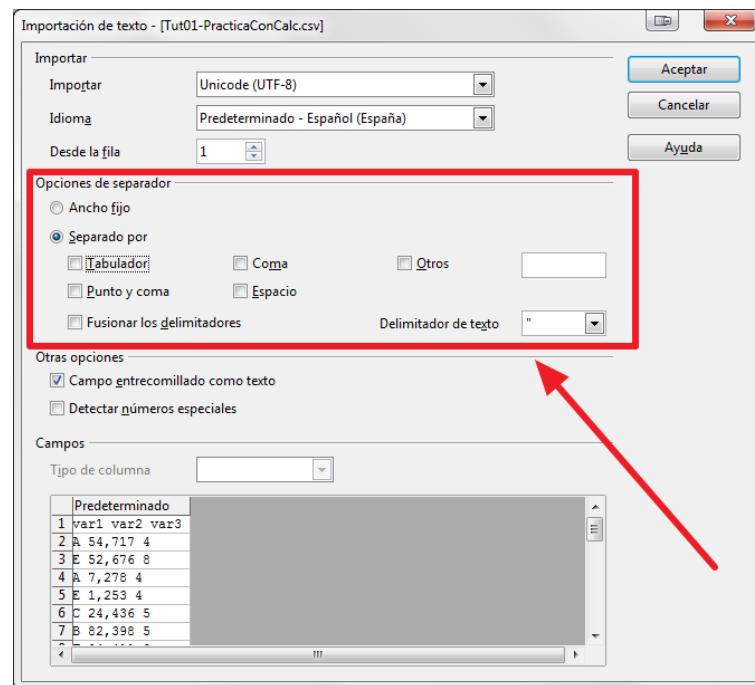
Es una excelente idea echarle un vistazo al fichero csv con un editor de texto antes de lanzarnos a hacer otras operaciones. Consideralo el primer paso de la descripción estadística de los datos, llamada también *Análisis Exploratorio de Datos*.

### 5.1. Abriendo el fichero con Calc.

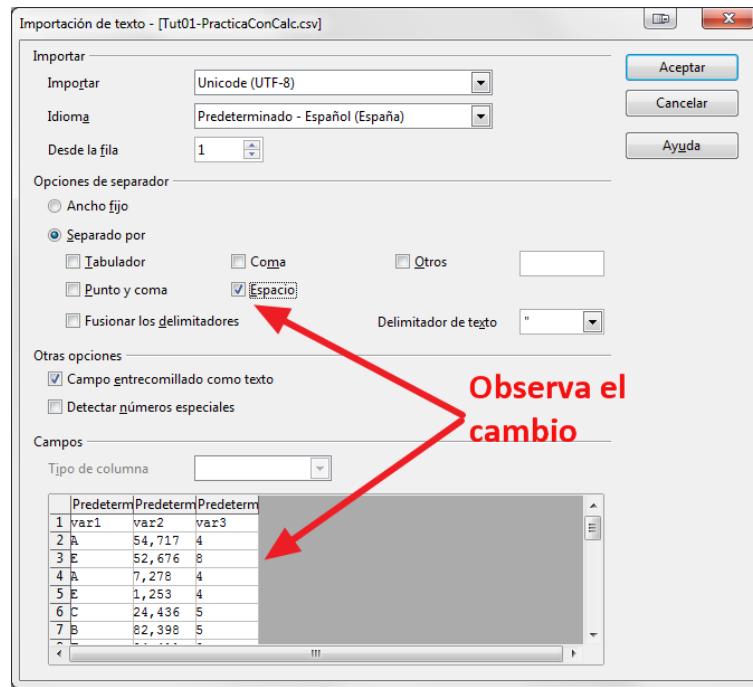
Si no lo has hecho, cierra el editor de texto en el que hemos abierto el fichero **csv**. Para seguir avanzado, vamos a abrirlo con la hoja de cálculo Calc. Una vez iniciado Calc, usa el menú **Archivo** → **Abrir** y navega hasta la carpeta **datos** de tu *Directorio de Trabajo* en la que has guardado el fichero **Tut01-PracticaConCalc.csv**. Cuando lo selecciones para abrir se debería abrir un cuadro de diálogo como el de la siguiente figura, que vamos a analizar:



Hemos indicado con una flecha roja la primera zona en la que debes fijarte. Calc te muestra una vista previa de su interpretación del fichero de datos. En el caso que se muestra en la figura, esa interpretación no coincide con lo que nosotros queremos obtener. Ten en cuenta que en tu ordenador las cosas pueden ser distintas, porque la interpretación de Calc depende de las opciones que se hayan seleccionado en la zona del cuadro de diálogo que hemos destacado en esta figura:



Aunque los ficheros **csv** deban su nombre a las comas, en realidad, se pueden usar (y se usan) distintos símbolos como **separadores** entre las distintas columnas de la tabla de datos que contiene el fichero. En los países que, como España, usan la coma como separador del punto decimal, es habitual usar un espacio, o un punto y coma, o un tabulador para separar entre sí las columnas. Esa parte del cuadro de diálogo nos deja seleccionar cuál (o cuáles, a veces son varios) de los símbolos posibles se deben interpretar como símbolos de separación entre columnas. En este ejemplo, las columnas están separadas por un espacio. Así que marcamos la casilla de la opción **Espacio**, nos aseguramos de que no haya seleccionada ninguna otra opción, y, como en esta figura, vemos en la vista previa que ahora Calc está interpretando los datos como queremos que lo haga.



Ahora podemos pulsar en **Aceptar**, y veremos como Calc nos muestra los datos, colocando correctamente las columnas de nuestra tabla de datos.

The screenshot shows the OpenOffice.org Calc spreadsheet with the data imported from 'Tut01-PrácticaConCalc.csv'. The data is organized into three columns: 'var1', 'var2', and 'var3'. The first row contains the column headers. The data consists of 26 rows, each with a unique identifier (1-26), a category code (A-F), a numerical value, and a count (e.g., 54,717, 4). The spreadsheet interface includes a toolbar, menu bar, and various status indicators at the bottom.

	Predeterm	Predeterm	Predeterm
1	var1	var2	var3
2	A	54,717	4
3	E	52,676	8
4	A	7,278	4
5	E	1,253	4
6	C	24,436	5
7	B	82,398	5
8	F	94,411	3
9	E	17,865	6
10	D	27,52	6
11	F	14,274	2
12	A	61,88	4
13	A	22,722	4
14	C	95,965	3
15	B	39,324	3
16	D	7,697	3
17	C	90,413	2
18	C	27,803	6
19	E	3,667	4
20	B	82,971	5
21	D	12,873	2
22	C	24,736	5
23	F	90,227	6
24	E	57,626	5
25	D	43,317	2
26	D	48,753	6

En el próximo tutorial empezaremos a trabajar con estos datos. Pero, antes de abandonar esta sección, queremos inaugurar una costumbre que nos va a acompañar en todos los tutoriales del curso. De vez en cuando te propondremos un ejercicio, para que puedas practicar lo que acabamos de aprender.

### Ejercicio 1:

- Trata de repetir los pasos anteriores, para abrir en Calc el fichero adjunto:

[Tut00-Ejercicio01a.csv](#)

Es recomendable empezar explorando el fichero con un editor de texto.

- ¿De qué tipo crees que son las variables de cada una de las columnas?

3. *El juego de las diferencias:* Trata de repetir los pasos anteriores para abrir en Calc el fichero adjunto:

[Tut00-Ejercicio01b.csv](#)

que contiene exactamente los mismos datos, pero con algunas modificaciones en la forma en la que se han codificado en el fichero. ¿Qué diferencias son esas?



## 5.2. Esquila de datos. Modificando ficheros csv con un editor de texto.

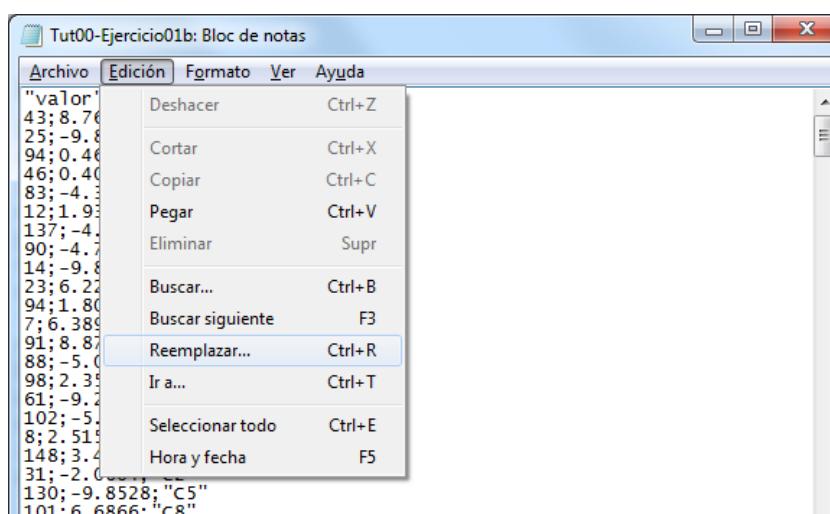
El fichero Tut00-Ejercicio01b.csv del Ejercicio 1 contiene una columna (la segunda, de nombre **medidas**), en la que se ha usado el punto, en lugar de la coma, como separador decimal. Eso puede suponer un problema para nosotros, porque algunos programas de ordenador usan la coma como separador decimal (por ejemplo, Calc en la versión en español), mientras que otros usan el punto (por ejemplo, R). Es frecuente, por tanto, encontrarse en la situación de tener que modificar un fichero de datos para cambiar puntos por comas, o viceversa. Esta es una operación típica (y sencilla) de lo que vamos a denominar **Esquila de Datos**. Es nuestra traducción del inglés *Data Wrangling*. Otra gente diría que están domando o domesticando datos, pero nosotros somos más de oveja, qué se le va a hacer.

Lo que tenemos que hacer, entonces, es cambiar los puntos por comas. Esta tarea, que en general consiste en reemplazar una cadena de texto por otra, la podemos acometer con un editor de texto sencillo como el *Bloc de Notas* de Windows. Vamos a dar los detalles para el *Bloc de Notas*, pero no deberías tener problemas en reproducirlos usando sus análogos en otros sistemas.

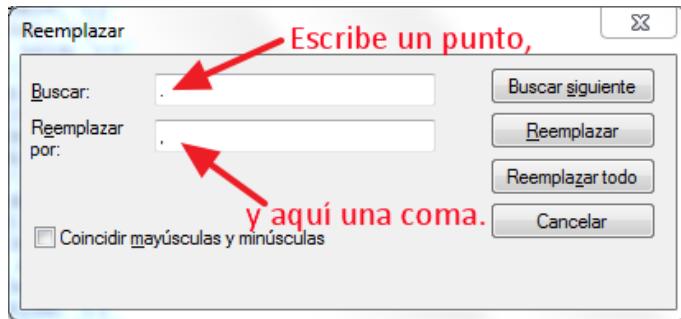
Al abrir el fichero Tut00-Ejercicio01b.csv con el *Bloc de Notas* veremos esto (sólo una parte del fichero resulta visible, dependiendo del tamaño de la ventana del editor en tu pantalla):

```
"valor";"medidas";"tipo"
43;8.7684;"C5"
25;-9.876;"C6"
94;0.46818;"C1"
46;0.40087;"C7"
83;-4.3824;"C2"
12;1.9344;"C3"
137;-4.5419;"C5"
90;-4.7231;"C8"
14;-9.8396;"C7"
23;6.2293;"C1"
94;1.8064;"C8"
7;6.3892;"C6"
91;8.8721;"C6"
88;-5.0858;"C4"
98;2.3532;"C2"
61;-9.2833;"C4"
102;-5.8757;"C1"
8;2.5154;"C4"
148;3.4106;"C5"
31;-2.0604;"C2"
130;-9.8528;"C5"
101;6.6866;"C8"
77;7.0499;"C3"
124;4.1762;"C4"
98;2.7039;"C4"
```

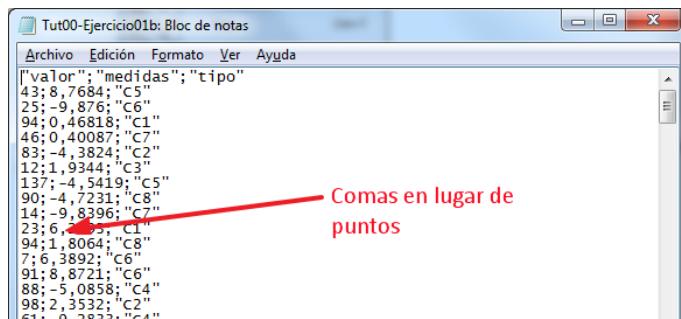
En el menú Edición, seleccionamos Reemplazar... (o pulsa **Ctrl+R**):



En el cuadro de diálogo que aparece escribe un punto en **Buscar** y una coma en **Reemplazar por**, como indica la figura:



Luego pulsa **Reemplazar todo**. Aunque el cuadro de diálogo no se cierra, los cambios ya se han hecho. Puedes cerrar ese cuadro de diálogo para verlo:



### Ejercicio 1:

Usando ese mismo fichero,

1. Reemplaza el separador de columnas (punto y coma) por el símbolo #.
2. Guarda el fichero modificado con el nombre **Tut00-Ejercicio01c.csv**, y ábrelo en *Calc*. Cuidado con las opciones de importación de ficheros **csv** en Calc, tendrás que usar la opción **Otros** para indicar el separador que estamos usando.
3. Para practicar un poco más el tema de los separadores y la importación de ficheros **csv**, aquí tienes el fichero adjunto:

[Tut00-Ejercicio01d.csv](#)

que puedes ver en la figura:

Fichero	Editar	Opciones	Codificación	Ayuda	23 %
valor	medidas	tipo			
21	0,2583	C7			
134	-8,967	C8			
116	6,141	C7			
8	5,197	C3			
104	-1,432	C3			
147	-6,055	C3			
128	-1,018	C2			
88	-4,107	C8			
149	7,074	C3			
84	0,5444	C8			
142	-2,369	C5			
115	-9,708	C3			
145	1,439	C7			
122	-6,816	C5			
74	-8,427	C5			
81	0,0697	C6			

Las columnas son más fáciles de reconocer a simple vista porque se han usado *tabuladores* como separadores entre columnas. Prueba a importar este fichero en Calc. Cuando lo hayas hecho, prueba a reemplazar los tabuladores por espacios (ábrelo en el *Bloc de Notas* y selecciona un tabulador con el ratón, para poder copiarlo y pegarlo en el cuadro de diálogo **Reemplazar**). Después, importa ese fichero modificado con Calc. Y, finalmente, cambia los separadores por comas, y repite el proceso de importación en Calc. ¿Hay algún problema?



## 6. Instalación de Anaconda Python. Python2 vs Python 3.

En esta colección de tutoriales vamos a usar , de forma prioritaria, el lenguaje de programación Python. La página oficial del lenguaje es

<https://www.python.org/>

Esa página contiene la documentación oficial de Python y la visitaremos varias veces en estos tutoriales. También puede utilizarse esta página para descargar las versiones oficiales del lenguaje. **¡Pero no lo hagas!** Nosotros vamos a utilizar una versión distinta, adaptada a nuestras necesidades

Lo primero que debes saber es que las versiones 2 y 3 de Python no son compatibles al 100 %. Algunos programas escritos usando Python 2 no funcionarán con Python 3 y viceversa. Por esa razón la primera decisión que debemos tomar es cuál es la versión de Python que vamos a usar:

En estos tutoriales vamos a usar siempre la versión 3 de Python.

Además de esa diferencia entre Python 2 y Python 3, tienes que saber que Python es lo que se conoce como un lenguaje de programación *de propósito múltiple* (en inglés *multipurpose*) o, si lo prefieres, *generalista* . Eso significa que Python se puede usar (y de hecho se usa) para actividades tan distintas como la computación científica, la creación de aplicaciones para servidor web, el procesamiento de lenguaje natural, etc. Cada una de esas actividades tiene sus propios objetivos y necesidades. Con el tiempo han ido surgiendo comunidades de usuarios centradas en torno a esas actividades. Esas comunidades diseñan las herramientas y los programas en Python que necesitan, y las comparten a través de internet. Naturalmente, si al instalar python tuviéramos necesariamente que instalar todas esas aportaciones el resultado sería un desperdicio de recursos y aumentarían mucho las dificultades de gestión del sistema; lo cual a su vez conlleva un aumento del número de errores. Los desarrolladores de Python han optado, como se hace en muchos otros lenguajes, por dotar al lenguaje de una estructura modular que permite a cada usuario instalar y utilizar sólo la parte del lenguaje que necesita. Eso ha contribuido a la aparición de lo que se conoce como **distribuciones** de Python. Una distribución lleva asociado un conjunto de herramientas para programar en Python, diseñadas para facilitar a los usuarios finales la instalación y mantenimiento de esas herramientas. A menudo las distribuciones tienen una cierta orientación temática. Por ejemplo, existen distribuciones orientadas a la computación científica y nosotros vamos a utilizar una de ellas. Concretamente, vamos a utilizar la distribución **Anaconda Python**, creada por la empresa Continuum Analytics. Entre las características destacables de esta distribución podemos señalar:

- Anaconda se puede instalar en sistemas Windows, Mac y Linux, tanto de 32 como de 64 bits, y proporciona una experiencia similar de trabajo en todos ellos.
- Esta distribución incluye el sistema **conda** de gestión de programas, que permite la fácil instalación de módulos adicionales. Además, también permite crear entornos virtuales de programación, con lo que por ejemplo resulta comparativamente sencillo alternar entre Python 2 y Python 3 en una misma máquina.
- Finalmente, Anaconda es una distribución orientada al Análisis de Datos, por lo que incluye herramientas de ese área temática, incluidas las de computación científica.

Pasando a las cuestiones prácticas, para instalar Anaconda debes usar el navegador de Internet para dirigirte a esta página:

<https://www.continuum.io/downloads>

Verás algo similar a lo que muestra la Figura 1. En el resto de esta sección vamos a describir la instalación de Anaconda en un sistema Windows, versiones 7/8/10. Si ese es tu caso, puedes pasar a los *Consejos generales*. Pero si usas Mac o Linux, te conviene leer el párrafo correspondiente aquí debajo, antes de seguir.

### Mac OS X.

Si tu sistema es Mac OS X no te preocupes, la instalación es bastante parecida, pero en ese caso lo primero que debes hacer es elegir tu sistema operativo en el enlace que hemos destacado en la parte superior de la Figura 1. Ese enlace te llevará a las instrucciones de instalación adecuadas para tu sistema. Si es un Mac, te recomiendo que uses la instalación gráfica.

## Linux.

Si usas Linux, puedes también instalar Anaconda, eligiendo el paquete de instalación adecuado para tu sistema. Pero para usuarios de distribuciones Linux basadas en Debian (como Ubuntu, Mint y otras distribuciones populares), existe otra opción, que pasa por utilizar paquetes nativos de la distribución. En la mayoría de los casos, basta con ejecutar

```
sudo apt-get install spyder3
```

Lo cual instalará todo el software necesario (después de pedir tu clave de administrador y siempre que tengas conexión a internet, claro). Si decides seguir este camino, una vez concluida la instalación puedes buscar el ícono para arrancar Spyder3 en tu sistema (asegúrate de que se trata de *Spyder3* y no *Spyder* sin el 3). También puedes abrir una terminal y ejecutar:

```
spyder3 &
```

Si usas otra distribución de Linux te recomiendo que busques información reciente en Internet, por ejemplo escribiendo en el buscador el nombre de tu distribución junto con la palabra **spyder3**.

**Consejos generales sobre la instalación (todos los sistemas).** En cualquier caso, si tienes problemas lo mejor es:

- Leer las instrucciones para Windows que aparecen a continuación, por si te sirven para aclarar tus dudas.
- Usar un buscador de Internet. Busca algo como “*Instalar Anaconda Python en ...*” y añade detalles de tu sistema. Asegúrate de que usas documentos recientes, de no más de un año de antigüedad a ser posible.
- Acude a tu ninja informático . En particular, si tu profesor te ha dicho que leas este tutorial en algún curso universitario, pídele que te ayude con la instalación.



Asegúrate también, antes de comenzar, de que tu equipo **dispone de espacio en disco suficiente para la instalación**. Anaconda Python es un programa relativamente grande, que ocupará más de 300Mb de espacio en tu disco en la versión Windows. Posiblemente más a medida que añadas módulos.

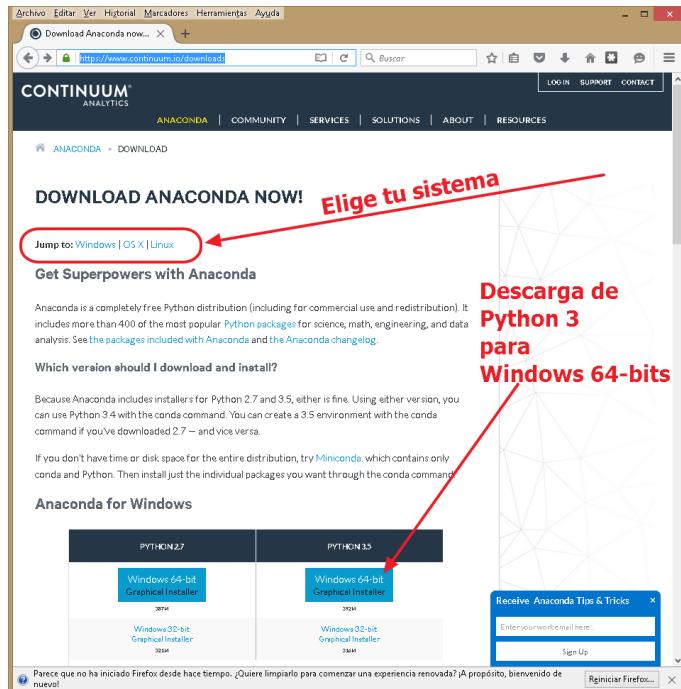


Figura 1: Página de descarga de Anaconda Pyhton.

Para instalar Anaconda en un sistema Windows de 64 bits debemos empezar por descargar el programa instalador haciendo clic en el enlace que hemos destacado con una flecha roja en la Figura 1. ¿Que no sabes si tu sistema es de 32 o de 64 bits? Busca el icono de *Este Equipo*, haz clic con el botón derecho y selecciona *Propiedades*. En la ventana que aparece verás varias propiedades de tu equipo y entre ellas el tipo de sistema.

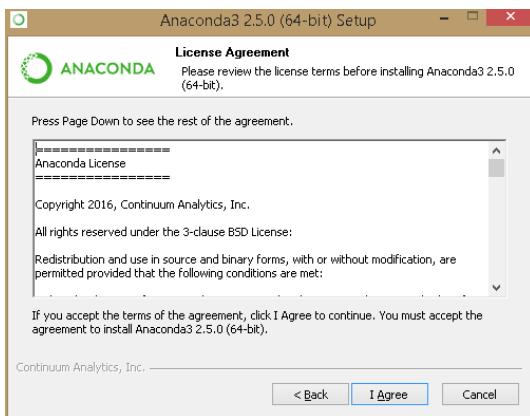
El programa instalado se llama algo parecido a *Anaconda3-2.5.0-Windows-x86\_64.exe*, aunque el número de versión puede haber variado cuando leas esto. Guarda ese fichero en algún lugar de tu ordenador y ejecútalo como harías con cualquier otro instalador de Windows. Veamos paso a paso el progreso de la instalación:

- La primera ventana del instalador es la que aparece en la Figura 2(a). Pulsa en *Next* para continuar la instalación.
- En la siguiente ventana (Figura 2(b)) tendrás que aceptar las condiciones de licencia de Anaconda.
- Ya en la tercera ventana (Figura 2(c)) aparece la **primera decisión relevante**: tenemos que elegir entre una instalación individual en el directorio del usuario que está ejecutando el instalador y una instalación global para todos los usuarios de la máquina. Si no sabes lo que tienes que elegir, lo mejor que puedes hacer es quedarte con la opción recomendada (seleccionada por defecto) y elegir una instalación individual. Si en el futuro tú o algún otro usuario del equipo necesita instalar otra versión de Python, esta opción es la que tiene menor riesgo de generar problemas de compatibilidad.
- A continuación (Figura 2(d)) el instalador nos ofrece la posibilidad de elegir el directorio en el que se va a instalar Python. Salvo que tengas buenas razones para hacer lo contrario, te aconsejamos que aceptes la opción que se ofrece y que instalará Anaconda Python en una carpeta de tu directorio personal llamada **Anaconda3**.
- En la siguiente pantalla (Figura 2(e)) tenemos que tomar **otra decisión que puede ser importante para algunos usuarios**. Si ya eras usuario de Python y tenías instalada otra distribución entonces seguramente querrás desmarcar las dos casillas que aparecen en esta ventana, para evitar que Anaconda interfiera con tu otra instalación de Python. Pero si esta es la única instalación de Python en tu equipo, lo mejor es que dejes ambas casillas marcadas, porque eso simplificará mucho nuestra relación con Anaconda Python en el futuro. En caso de duda, aplica los remedios habituales: busca, pregunta, infórmate.
- Tras pulsar *Next* una vez más comenzará el proceso de instalación propiamente dicho (Figura 2(f)), en el que una barra de progreso te mostrará como se van copiando los ficheros de instalación de Anaconda Python. Esperamos a que se complete ese proceso (Figura 2(g)) y hacemos clic en *Next*.
- La última ventana (Figura 2(h)) nos informa de que Anaconda Python ya está instalado hacemos clic en *Finish* para salir de la instalación.

(a)



(b)



(c)

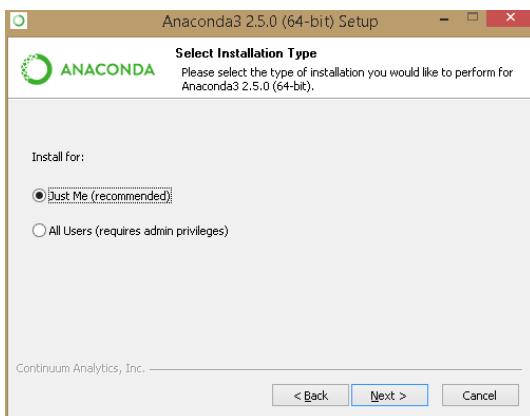
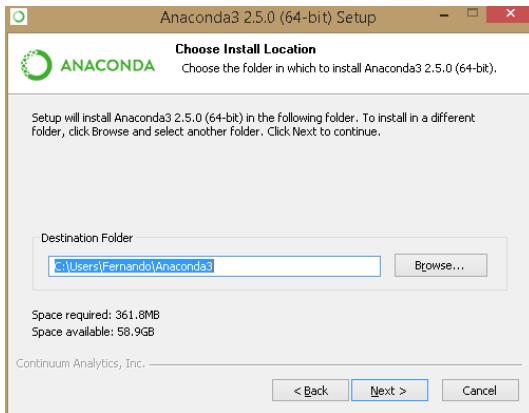
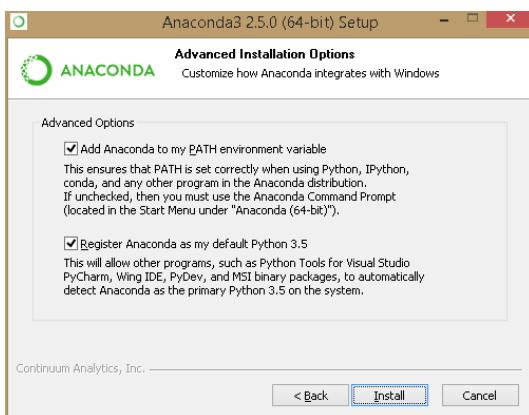


Figura 2: Primeras etapas de la instalación de Anaconda Python.

(d)



(e)



(f)

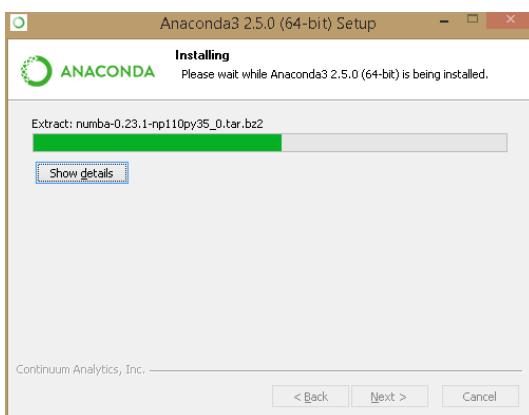
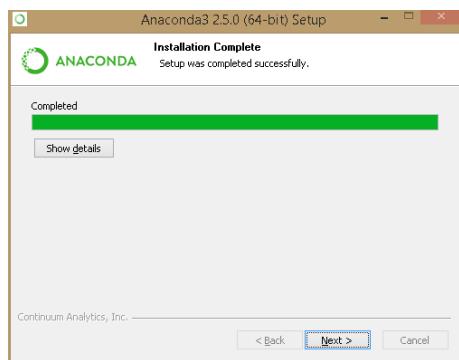
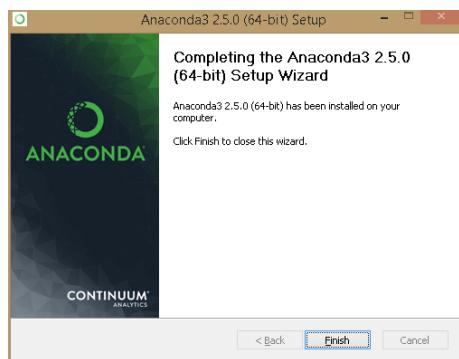


Figura 2, continuación. Etapas intermedias de la instalación de Anaconda Python.

(g)



(h)



(i)

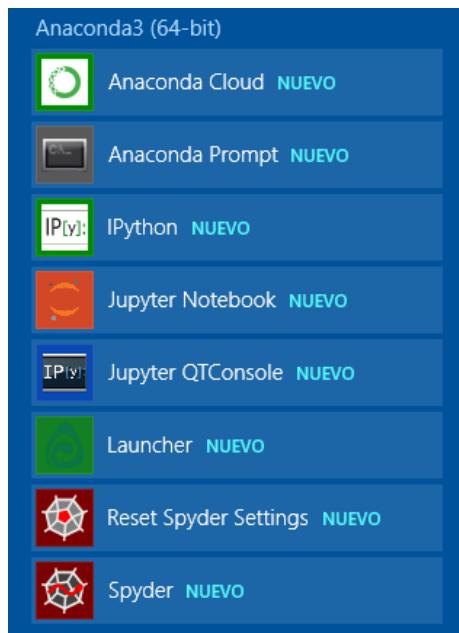


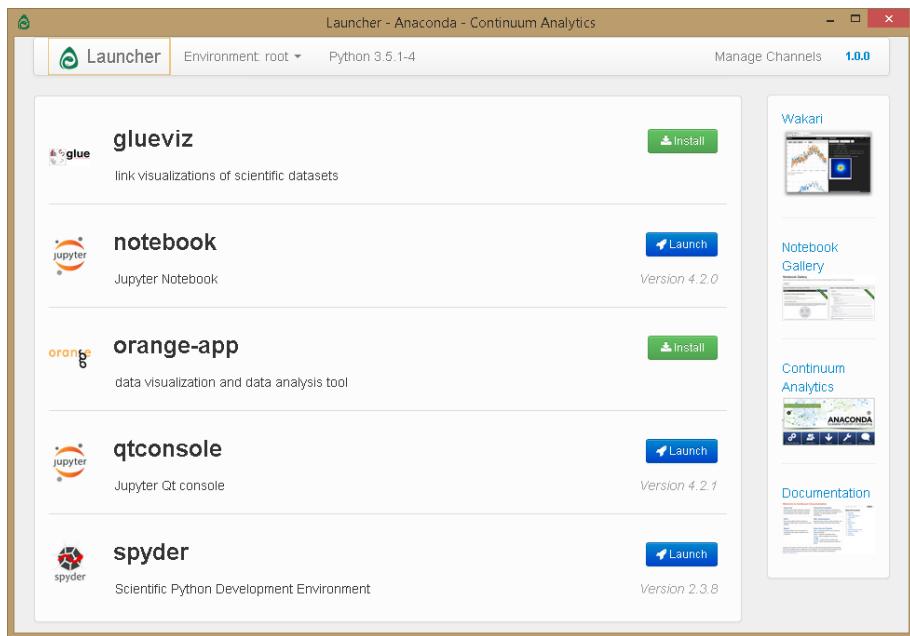
Figura 2, continuación. Etapas finales de la instalación de Anaconda Python.

## 6.1. Iniciando Spyder.

En Windows esta instalación creará un grupo de programas llamado *Anaconda3* en tu equipo, que puedes localizar en la lista de *Aplicaciones*. La ubicación de esa lista es distinta en cada una de las últimas versiones de *Windows*. Si tienes problemas para localizarla pide ayuda a alguien con más experiencia. Una vez localizado el grupo de programas *Anaconda3*, verás los iconos de varios programas, algunos de los cuales exploraremos en futuros tutoriales. Pero una de las ventajas de usar Anaconda es que disponemos de una especie de *centro de control* de esos programas, llamado *Launcher*, que nos permite, entre otras cosas, mantenerlos actualizados fácilmente. Puedes iniciar *Launcher* mediante el ícono:



En Mac OS X, tras instalar Anaconda puedes encontrar el ícono de *Launcher* en el *Launchpad*. En cualquier caso, al iniciar *Launcher* verás una ventana parecida a esta:



en la que aparecen los iconos de algunos componentes de Anaconda que puedes iniciar desde esta ventana. Más adelante, cuando llegues a convertirte en un usuario experimentado de Python, es posible que quieras utilizar algunas características más avanzadas. Por ejemplo, para poder usar Python2 y Python3 simultáneamente en el mismo ordenador. En ese momento te aconsejamos que dediques algún tiempo a estudiar la documentación de Anaconda, porque Launcher puede simplificar mucho tu trabajo.

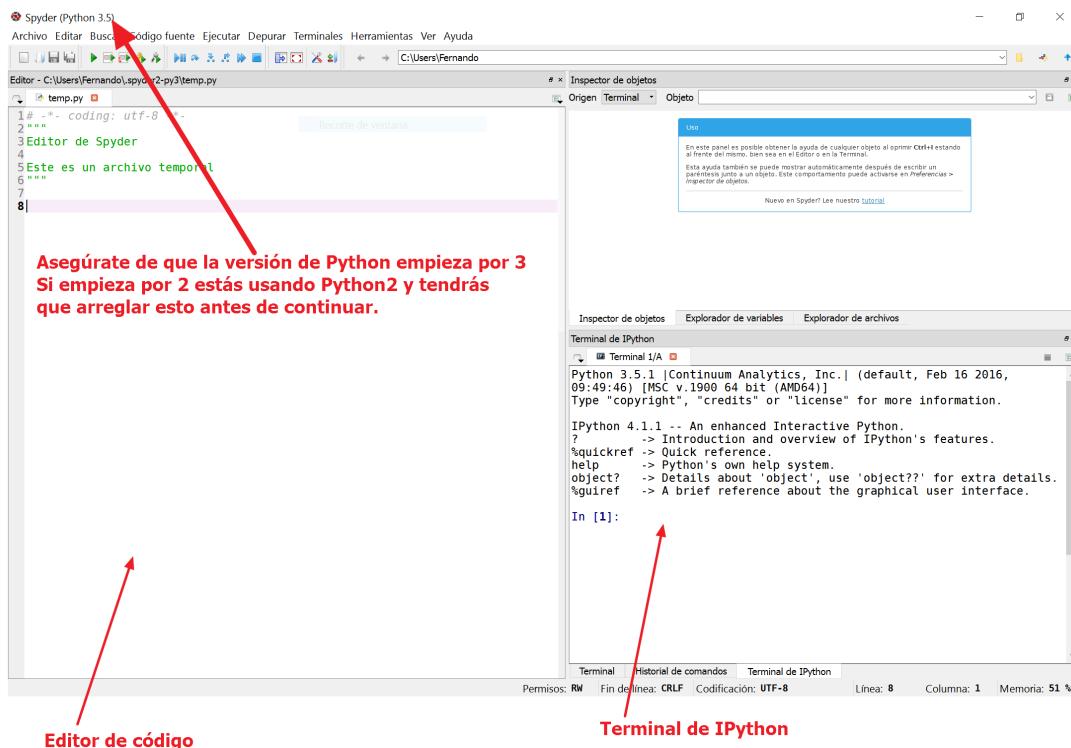
Nosotros vamos a usar esta ventana para acceder a *Spyder*. Si tu sistema operativo es Linux (Debian/Ubuntu/Mint) y optaste por instalar paquetes nativos, tú no dispondrás de Launcher. Pero recuerda que puedes iniciar el programa ejecutando en una terminal el comando:

```
spyder3 &
```

Volviendo a los usuarios de *Launcher* en Anaconda, si esta es la primera vez que lo inicias es muy posible que tengas que instalar el programa *Spyder* (necesitarás estar conectado a Internet). Si es así, a la derecha del ícono de Spyder aparecerá un botón verde con el texto *Install*. Pulsa ese botón y espera a que termine la instalación de Spyder. Una vez instalado, el botón cambiará al color azul con el texto *Launch*. Como hemos dicho antes, una de las ventajas que aporta *Launcher* es que facilita la tarea de mantener actualizada nuestra instalación de Python. Anaconda comprobará periódicamente si existen actualizaciones de Spyder y otros componentes. Si es así, verás aparecer

un botón adicional con el texto *Update*, que al pulsarlo actualizará ese programa. Te aconsejamos que mantengas actualizado tu instalación de Python (y en general todo tu sistema).

Cuando *Spyder* esté instalado, arráncalo haciendo click sobre el botón azul *Launch*. Ten un poco de paciencia: puede tardar un poco, especialmente la primera vez. La ventana inicial de Spyder, en cualquier caso, será parecida a esta:



Hemos indicado en la figura los dos paneles de la ventana de Spyder en los que se va a desarrollar de manera fundamental nuestro trabajo, el **Editor de Código** y la **Terminal de IPython**. Comprueba además la versión de Python en la barra superior de la ventana de Spyder, como se indica en la figura.

### ¿Qué es Spyder?

En los próximos tutoriales tendremos ocasión de conocer a fondo el programa. Por el momento nos vamos a tener que conformar con una brevíssima introducción. El nombre Spyder es un acrónimo de **S**cientific **P**Ython **D**evelopment **E**nviRonment. Es decir, un *entorno de desarrollo integrado* para Python con una orientación científica. En la página web

<https://pythonhosted.org/spyder/>

puedes encontrar más documentación sobre Spyder. ¿Y qué es un entorno de desarrollo integrado? Es una herramienta típica del trabajo de programación. Al principio los programadores usaban varias herramientas separadas para su trabajo: un editor de texto, un compilador, un sistema de control de versiones, etc. Los entornos de desarrollo integrado surgieron de modo natural como una manera de reunir, en un único lugar, casi todas las herramientas que el programador necesitaba para su trabajo, aprovechando además las sinergias que se crean al coordinar de esa forma el trabajo. Dicho eso, conviene añadir enseguida que muchos programadores prefieren todavía trabajar *al modo clásico*, con herramientas separadas para cada parte del trabajo. Es cuestión de gustos y del análisis que cada programador hace de las herramientas que le hacen más productivo. En este curso aprenderemos a usar Python mediante Spyder, pero la decisión final sobre la forma en la que usas el lenguaje será tuya.

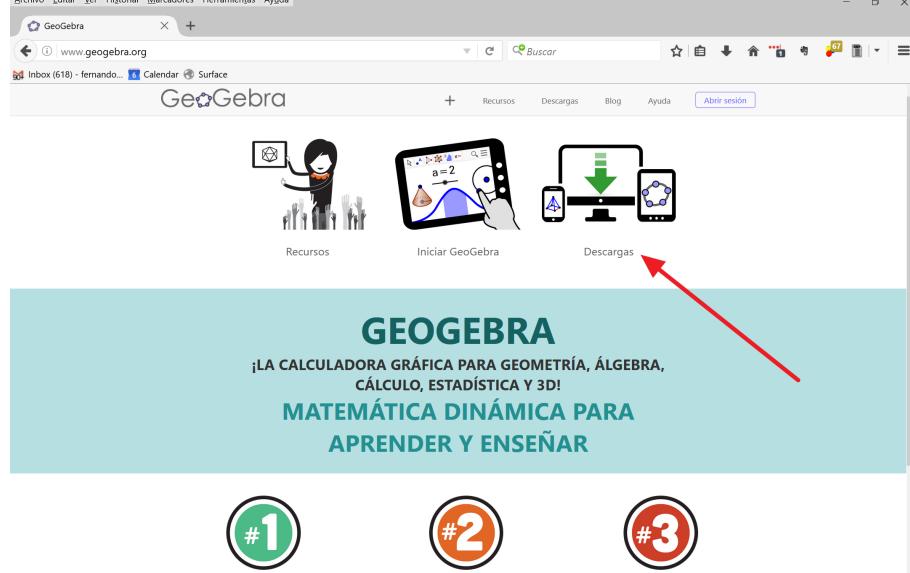
## 7. Instalación de GeoGebra.

GeoGebra es un programa gratuito y de código abierto, que, según sus creadores, permite la *interacción dinámica de geometría, álgebra, estadísticas y recursos de análisis y cálculo*. GeoGebra

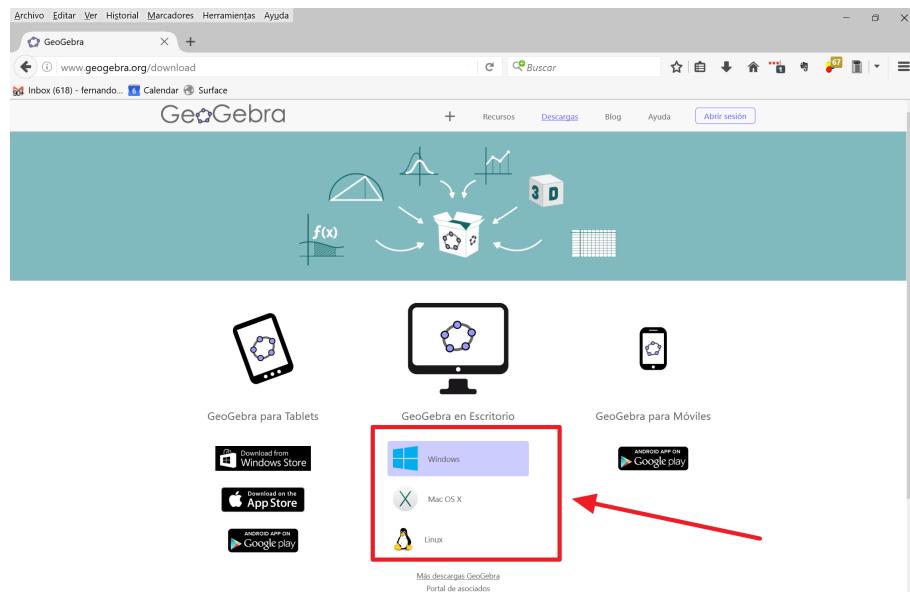
se diseñó para servir de apoyo visual a la enseñanza de las matemáticas, y en cada nueva versión ha ido aumentando sus capacidades. En particular, para lo que aquí nos interesa, GeoGebra ofrece bastantes herramientas para trabajar con distribuciones de probabilidad, y algunas operaciones básicas de la Estadística. En este curso vamos a usar GeoGebra sobre todo para mostrar algunas construcciones dinámicas, en las que podrás interactuar con algunos elementos de la construcción, para experimentar lo que sucede cuando se modifican.

La página principal del proyecto GeoGebra, en la que puedes encontrar mucha información sobre el programa es:

[www.geogebra.org](http://www.geogebra.org)



En esa página, pulsa sobre el enlace Descargas que hemos destacado en la anterior figura.



y elige tu sistema en la ventana que se abre. La descarga del instalador debería comenzar en ese momento. A partir de aquí, las instrucciones de instalación que incluimos son para el sistema Windows. Tras ejecutar el instalador pasarás por estas pantallas:



## Welcome to the GeoGebra 5 Setup Wizard

Before beginning the installation of GeoGebra 5, please choose a language:

Presione Siguiente para continuar.

Language:

Español

GeoGebra 5.0.232.0 (April 28 2016)

Siguiente >

Cancelar

Pulsamos en Siguiente

### Acuerdo de licencia

Por favor revise los términos de la licencia antes de instalar GeoGebra 5.

Presione Avanzar Página para ver el resto del acuerdo.

GeoGebra - Dynamic Mathematics for Everyone  
<http://www.geogebra.org/>

#### LICENSE

You are free to copy, distribute and transmit GeoGebra free of charge for non-commercial purposes (see conditions and details below).

#### PROJECT DIRECTOR

\* Markus Hohenwarter (Austria & USA 2001-)

#### LEAD DEVELOPER

\* Michael Borchert (UK 2007-)

#### DEVELOPERS

\* Gabor Acsin (Hungary 2009-)

\* Balazs Bencze (Romania 2012-)

\* Mathieu Blossier (France 2008-)

\* Arnaud Delobelle (UK 2011-)

\* Calixte Denizet (France 2010-)

\* Judit Elias (Hungary 2009-)

\* Arpad Fekete (Hungary 2010-)

\* Laszlo Gal (Hungary 2013-)

\* Zbynek Konecny (Czech Republic 2010-)

\* Zoltan Kovacs (Hungary 2010-)

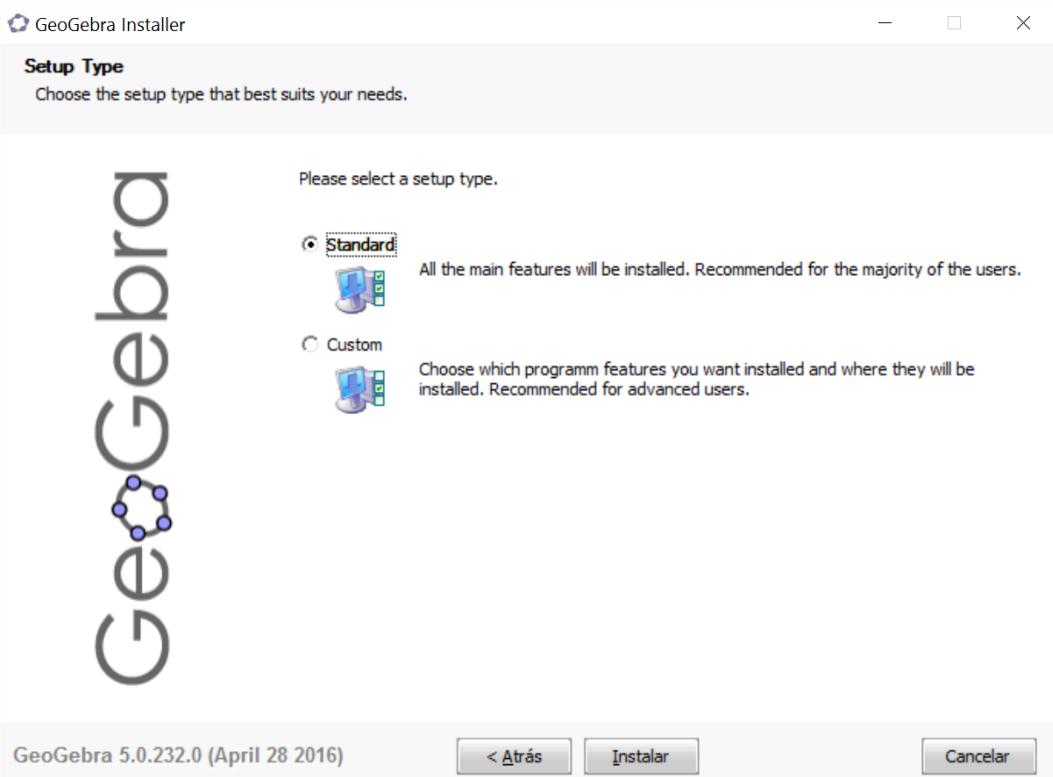
Si acepta todos los términos del acuerdo, seleccione Acepto para continuar. Debe aceptar el acuerdo para instalar GeoGebra 5.

GeoGebra 5.0.232.0 (April 28 2016)

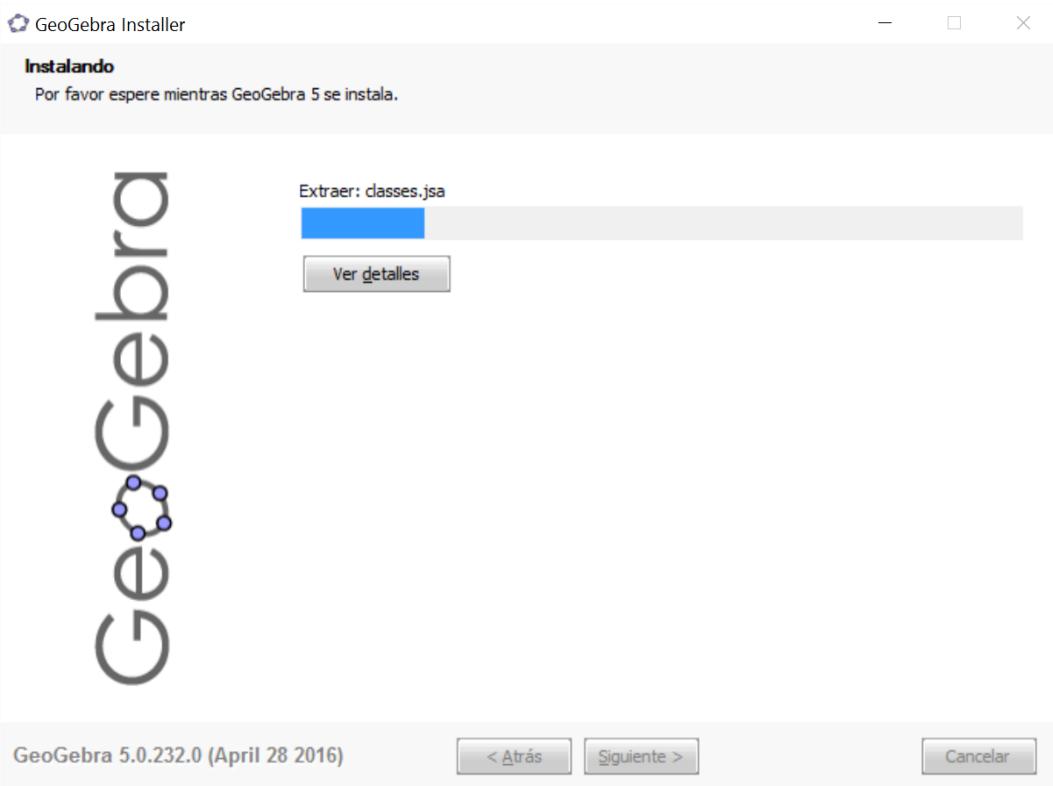
Acepto

Cancelar

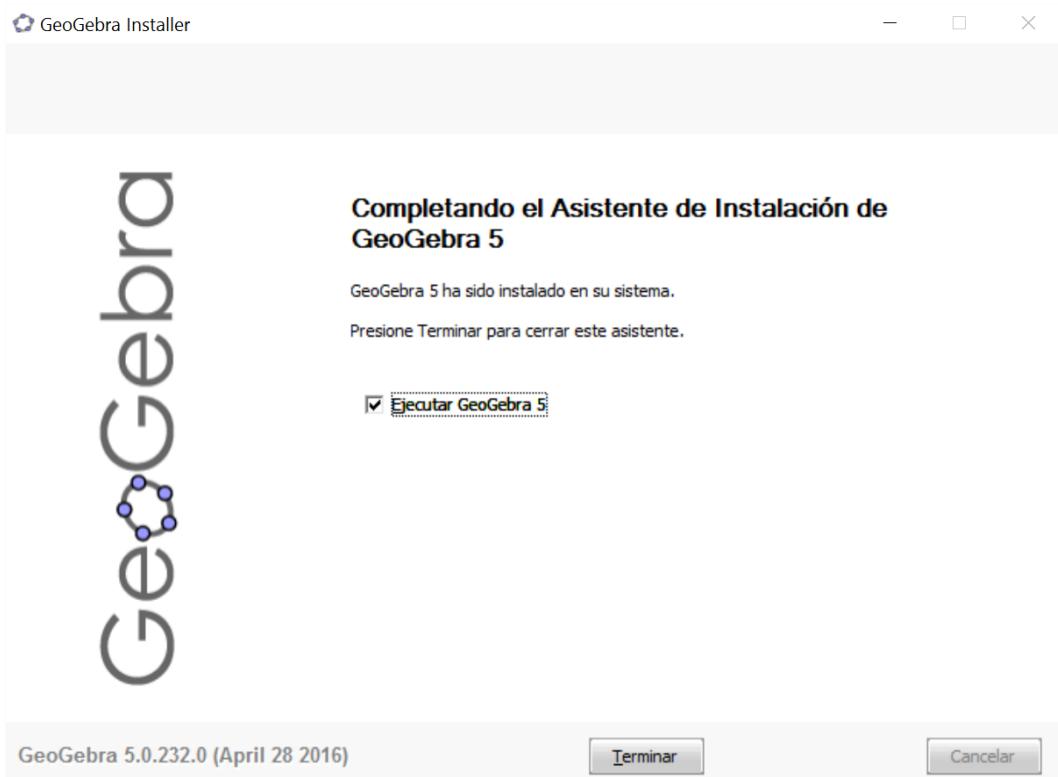
Pulsamos en Acepto



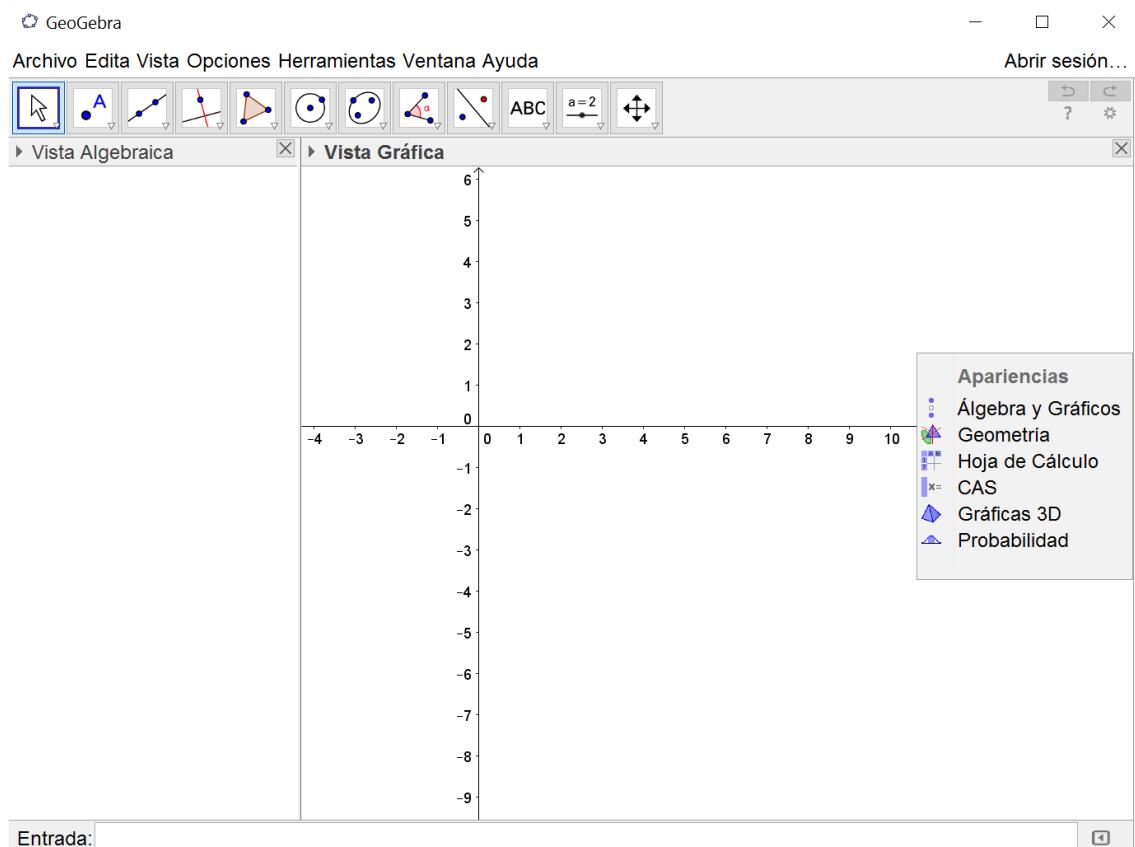
Puedes dejar la instalación Standard seleccionada, y pulsar en Instalar:



Esperamos unos momentos mientras se instala el programa ...



...y la instalación concluye correctamente. Para comprobar que ha sido así, deja marcada la casilla Ejecutar GeoGebra y pulsa en Terminar. Al cabo de unos instantes aparecerá la interfaz de GeoGebra, que se muestra en esta figura:



Como ves, la mayor parte la ocupa la **Vista Gráfica**, en la que aparecen los ejes de un plano de coordenadas cartesianas. Justo debajo aparece la *Línea de Entrada*, que usaremos para teclear comandos. En este curso no vamos a profundizar en el uso de GeoGebra. Vamos a usarlo para visualizar construcciones que te entregaremos adjuntas en los capítulos de teoría o en los tutoriales. Así que podrás usarlas directamente, y ya verás que resultan muy intuitivas. También usaremos la

*Calculadora de Probabilidades* y la *Ventana de Cálculo Simbólico*, dos herramientas de GeoGebra que facilitarán mucho nuestro trabajo. Pero no vamos a explorar, ni mucho menos, todas las posibilidades que ofrece el programa. En cualquier caso, si quieres aprender más sobre GeoGebra (que es un gran programa para la enseñanza y la visualización de las Matemáticas), te recomendamos que explores su página web.

## 8. Siguiente paso. ¿Dónde vamos ahora?

Tras instalar todo este software, hay que ponerlo a trabajar. En general, como hemos dicho en la Introducción del libro, cada capítulo del libro se corresponde con un tutorial, y la numeración de capítulos y tutoriales coincide. Sin embargo, los Tutoriales 1 y 2, que corresponden a la Parte I del curso, son especiales. Cada uno de ellos cubre el contenido conjunto de los Capítulos 1 y 2 de esa parte del curso. Pero en el Tutorial01 se utiliza la hoja de cálculo Calc de OpenOffice, mientras que en el Tutorial02 empezaremos a usar Python.

En el resto del curso, cada pareja Capítulo/Tutorial vendrá acompañada de una *Guía de Trabajo*, un documento breve que esencialmente explica como se coordina el trabajo teórico del capítulo con los contenidos prácticos del tutorial. De nuevo, los dos primeros capítulos y tutoriales son un caso especial, porque en este caso existe una única *Guía de Trabajo* conjunta para ambos. Y ese es el siguiente paso: debes abrir ese documento y seguir sus instrucciones. El documento estará disponible en la página web del libro, o de la forma que te indique tu profesor. Las *Guías de Trabajo* constituirán el guión que ordene nuestro trabajo en el curso.

---

Fin del Tutorial-00. ¡Gracias por la atención!

## Tutorial 02: Estadística descriptiva con Python.

Atención:

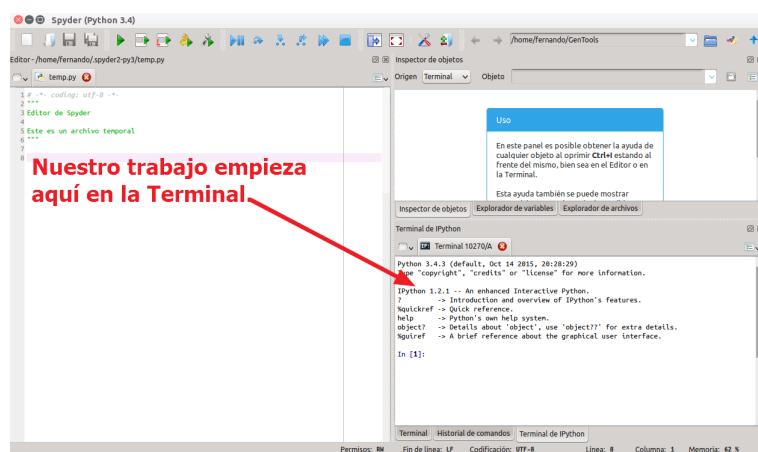
- Este documento pdf lleva adjuntos algunos de los ficheros de datos necesarios. Y está pensado para trabajar con él directamente en tu ordenador. Al usarlo en la pantalla, si es necesario, puedes aumentar alguna de las figuras para ver los detalles. Antes de imprimirla, piensa si es necesario. Los árboles y nosotros te lo agradeceremos.
- Fecha: 15 de mayo de 2016. Si este fichero tiene más de un año, puede resultar obsoleto. Busca si existe una versión más reciente.

## Índice

<b>1. Primer contacto con Python.</b>	<b>1</b>
<b>2. Variables y listas en Python.</b>	<b>13</b>
<b>3. El editor de código y ficheros de código Python.</b>	<b>26</b>
<b>4. Bucles for.</b>	<b>34</b>
<b>5. Recursos para facilitar el trabajo: ordenación y print con formato.</b>	<b>40</b>
<b>6. Comentarios.</b>	<b>46</b>
<b>7. Ficheros csv en Python.</b>	<b>50</b>
<b>8. Estadística descriptiva de una variable cuantitativa discreta con datos no agrupados.</b>	<b>54</b>
<b>9. Más operaciones con listas.</b>	<b>65</b>
<b>10. Instrucción if y valores booleanos.</b>	<b>73</b>
<b>11. Ejercicios adicionales y soluciones.</b>	<b>79</b>

## 1. Primer contacto con Python.

Vamos a empezar arrancando Spyder, como aprendimos a hacer en el Tutorial-00 (recuerda que si usas Anaconda debes arrancar primero el Launcher). Al hacerlo te encontrarás con una ventana como esta:



La versión de Spyder que aparece en esa figura es la de Linux (Ubuntu), pero si usas otro sistema la ventana será muy parecida. Esas son una de las primeras ventajas de Spyder: una vez que un programador se acostumbra a usarlo, si tiene que cambiar de sistema operativo la adaptación resulta más fácil.

Nuestro trabajo va a empezar, como hemos indicado en la figura, en el panel de Spyder llamado *Terminal*. Las primeras líneas muestran información sobre la versión de Python que estamos usando, y sugieren algunas formas de obtener ayuda. Más adelante volveremos sobre algunas de ellas. Por el momento, la línea que más nos interesa es la última, en la que aparece:

In [1] :

El símbolo In [1] : es el *prompt* de Python. La palabra In indica que Python está esperando que teclees un comando (*entrada de comandos*) y el [1] indica que este será el primer comando de nuestra sesión de trabajo en Python. Haz click con el ratón a la derecha de esa línea, hasta que veas el cursor parpadeando. En ese momento Python está esperando para empezar a dialogar con nosotros. Empecemos: prueba a teclear  $2 + 3$  y pulsar *Enter*. El resultado será el que se muestra en el rectángulo gris justo aquí debajo:

```
In [1]: 2 + 3  
Out[1]: 5
```

In [2] :

Como ves, Python ha contestado inmediatamente debajo de la primera línea de entrada, añadiendo una *línea de salida*:

```
Out[1]: 5
```

con el resultado de la suma y el mismo número entre corchetes. Más adelante aprenderemos la utilidad de estos números. Además, y para que podamos seguir trabajando, la *Terminal* de Python muestra la siguiente línea de entrada

In [2] :

en la que el cursor parpadea, a la espera de nuestra siguiente instrucción. Por cierto, los espacios en blanco entre los números y el símbolo de operación + son irrelevantes. Se obtiene lo mismo si usas  $2+3$  en lugar de  $2 + 3$ . Una de las ventajas de esta propiedad de los espacios es que podemos usarla para hacer más legible el código que escribimos. Más adelante veremos otros casos en los que, por contra, los espacios son fundamentales. Pero no te preocupes, es fácil aprender a distinguir esos casos.

A parte de sumas podemos hacer, naturalmente, multiplicaciones y divisiones. Prueba a ejecutar la instrucción del siguiente rectángulo gris. Puedes copiar y pegar directamente desde aquí a la *Terminal* de Python:

```
6 * 5
```

Python usa el asterisco para representar la multiplicación, así que el resultado es 30. De la misma forma, Python usa la barra / para indicar la división. Compruébalo ejecutando:

```
16 / 5
```

El resultado 3.2 de la división muestra la forma en la que Python escribe los decimales, con punto separando la parte entera y la parte decimal. Puesto que estamos usando la versión 3 de Python, el resultado de una división hecha con / es siempre un número decimal. Prueba la siguiente operación:

```
16 / 4
```

Si alguna vez tienes que usar la versión 2 de Python descubrirás que allí la división usando / es un poco más complicada. Por el momento, seguiremos adelante sin preocuparnos de esto. Más adelante daremos más detalles.

Vamos a elevar un número al cubo. En muchos lenguajes de programación las potencias se indican con el símbolo  $\wedge$  (el acento circunflejo). Pero Python utiliza dos asteriscos: \*\*. Prueba a ejecutar esta operación:

Por cierto y aunque esto es una manía personal, fíjate en que en todos los casos hemos usado espacios entre el operador y los números (operando), salvo precisamente en este último caso de la potencia. La razón para trabajar así es porque creo que de esa forma el código resulta más legible.

### División entera: resto y cociente.

En alguna ocasión tendremos necesidad de calcular el cociente y el resto de una división entera. Por ejemplo, para convertir un tiempo en segundos al formato minutos-segundos. En Python 3 el cociente se obtiene con `//` y el resto con `%`. Así, un experimento que dura 475 segundos, expresado en el formato minutos-segundos dura:

```
In [3]: 475 // 60
Out[3]: 7
```

```
In [4]: 475 % 60
Out[4]: 55
```

Es decir, que el experimento dura 55 minutos y 7 segundos.

### Sobre la numeración del prompt de Python en estos tutoriales.

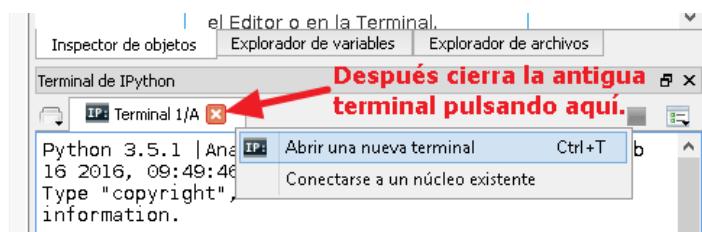
Es posible que te haya sorprendido ver el prompt

```
In [3]:
```

en el fragmento anterior de código. Si has ido ejecutando cada uno de los fragmentos de código que hemos visto hasta ahora tú debes estar viendo números mayores que 3. Pero el número concreto que aparece en la terminal no es importante y, desde luego, no influye en la respuesta de Python. Así que queremos aprovechar este momento para señalar, antes de seguir adelante, que no debes preocuparte si los números de línea en nuestros ejemplos no coinciden con los que aparecen en tu terminal de Python.

### Detalles adicionales sobre la terminal de Python en Spyder.

En raras ocasiones la terminal no responde, o se queda en blanco sin que veamos el prompt, etc. En esos casos te recomiendo que abras una nueva terminal, pulsando `Ctrl + T` (en Mac OS X, usa `⌘ + T`) o haciendo click con el botón derecho del ratón en la parte superior de la pestaña de la terminal, como ilustra esta figura, y seleccionando la opción correspondiente. No olvides cerrar la antigua terminal antes de seguir trabajando.



Otro problema que puede estar empezando a aparecer en tu trabajo es el tamaño de la terminal de Python en Spyder. Puesto que Spyder reparte el espacio de su ventana entre varios paneles, puede que el panel de terminal te resulte demasiado pequeño para ser cómodo (especialmente si trabajas en un ordenador portátil con una pantalla no demasiado grande). Y aunque puedes desplazarte arriba y abajo en el terminal con las flechas de desplazamiento del teclado (o con la rueda del ratón) hay un remedio más sencillo y cómodo. Haz clic con el ratón en este icono de la barra de herramientas de Spyder:



Al hacerlo verás que el panel de la terminal se expande hasta llenar casi toda la ventana de Spyder.

The screenshot shows the Spyder Python 3.4.4 interface. In the top bar, it says "Spyder (Python 3.4)". Below the menu bar, there's a toolbar with icons for file operations like Open, Save, Run, and Help. The main area is a "Console 0/A" window displaying Python code and its output. The code includes basic arithmetic operations (e.g., 2 + 3, 6 \* 5, 16 / 5, 16 // 4, 3\*\*2), division assignment (475 // 60), modulus (475 % 60), and integer division assignment (475 // 55). At the bottom of the console window, there are status bars for "Permissions: RW", "End-of-lines: LF", "Encoding: UTF-8-GUESSED", "Line: 12", "Column: 5", and "Memory".

```

Python 3.4.4 |Continuum Analytics, Inc.| (default, Jan 9 2016, 17:30:09)
Type "copyright", "credits" or "license" for more information.

IPython 4.1.2 -- An enhanced Interactive Python.
?            -> Introduce yourself to IPython's features.
%quickref  -> Quick reference.
%help       -> Python's own help system.
%object?   -> Details about 'object', use "%object???" for extra details.
%gui?      -> A brief reference about the graphical user interface.

In [1]: 2 + 3
Out[1]: 5

In [2]: 6 * 5
Out[2]: 30

In [3]: 16 / 5
Out[3]: 3.2

In [4]: 16 / 4
Out[4]: 4.0

In [5]: 3**2
Out[5]: 9

In [6]: 475 // 60
Out[6]: 7

In [7]: 475 % 60
Out[7]: 55

In [8]:

```

El proceso es reversible: otro click en el mismo ícono y vuelves a la situación anterior. Y además, aunque por el momento no los estamos usando, puedes aplicar la misma idea a los otros paneles de Spyder.

## 1.1. Funciones matemáticas y módulos de Python.

Aparte de las cuatro operaciones aritméticas básicas, las calculadoras científicas de mano incluyen funciones como la raíz cuadrada, logaritmos, funciones trigonométricas, etc. En Python, por supuesto, también podemos calcular esas funciones. Pero antes de hacerlo tenemos que aprender algo sobre el sistema de módulos de Python.

Siendo un lenguaje de programación de *propósito general* (es decir, no especializado en una tarea muy concreta), Python se puede utilizar para una gran cantidad de cosas distintas, cada una con sus necesidades específicas. Los programadores de Python han desarrollado muchísimas herramientas orientadas al cálculo científico, pero también a operaciones financieras, gestión de servidores web y sistemas, trabajo con bases de datos, etc. Si cada vez que usamos Python tuviéramos que instalar todo ese código estaríamos desperdiando una enorme cantidad de recursos y haciendo que el resultado fuera poco eficiente. Al fin y al cabo el usuario que quiere utilizar Python para Genómica no necesitará, seguramente, las funciones financieras más especializadas de Python (al menos, no a la vez). Por eso, como sucede en otros lenguajes, Python está organizado en una estructura modular, de manera que en cada momento podemos disponer de aquellas partes del código de Python que necesitamos. Concretamente, el código se organiza en **módulos** que puedes imaginar como cajas de herramientas. Y cuando queremos utilizar una herramienta concreta debemos indicárselo a Python pidiéndole que importe el módulo que contiene esa herramienta.

Veamos un ejemplo. Muchas funciones matemáticas, como la raíz cuadrada y otras que mencionábamos antes, están guardadas en un módulo (caja de herramientas) llamado **math**. La función raíz cuadrada se llama **sqrt** (del inglés *square root*). Para importar la función **sqrt** del módulo **math** en Python usamos este comando:

```
from math import sqrt
```

Cuando lo ejecutes aparentemente no pasará nada, verás algo como lo que se muestra a continuación. Recuerda que los números entre corchetes pueden ser distintos en la *Terminal* de Python (y que esta observación vale para todos los fragmentos de código que aparezcan a partir de ahora en los tutoriales):

```
In [5]: from math import sqrt

In [6]:
```

Hemos incluido la siguiente línea de entrada para que veas que debajo de **In [5]** no hay un **Out [5]**, no existe la correspondiente línea de salida. Pero aún así la instrucción ha hecho efecto: esa instrucción es nuestra forma de decir, en lenguaje Python, “*saca la herramienta sqrt de la caja math*”. Una vez hecho esto, podemos usar la herramienta para, por ejemplo, calcular la raíz cuadrada de 9 o cualquier otra. Compruébalo ejecutando esta operación:

```
sqrt(9)
```

y también esta:

```
sqrt(17)
```

Como ves, la función se ejecuta o *invoca* colocando su argumento entre paréntesis. El módulo `math` contiene muchas otras funciones además de la raíz cuadrada. Por ejemplo, las funciones trigonométricas seno, coseno y tangente que se representan, respectivamente mediante `sin`, `cos` y `tan`. Por defecto, esas funciones asumen que los ángulos se miden en radianes. Por su parte, el logaritmo natural (o neperiano) de base  $e$  se llama en Python `log` y la exponencial (para calcular  $e$  elevado a un número) se llama `exp`.

Pero, además de funciones, a menudo los módulos de Python contienen otro tipo de *objetos*. Por ejemplo, el módulo `math` contiene valores aproximados de las constantes matemáticas  $\pi$  y  $e$ , entre otras.

Recuerda que para usar estas herramientas las debemos empezar por importarlas. Para hacer más cómodo nuestro trabajo, podemos importar varias funciones de una vez:

```
from math import sin, cos, tan, log, exp, pi, e
```

Una vez hecho esto podemos empezar a usar las funciones. Por ejemplo, en trigonometría elemental hemos aprendido que:

$$\sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2} \approx 0.7071$$

Vamos a calcular con Python este número de dos formas. Ejecuta primero esta versión :

```
sin(pi/4)
```

cuyo resultado es:

```
0.7071067811865475
```

Y después ejecuta:

```
sqrt(2)/2
```

cuyo resultado es:

```
0.7071067811865476
```

¡Fíjate en que las últimas cifras son distintas! Eso nos debe servir de recordatorio de que los cálculos que estamos realizando son *aproximaciones numéricas*, no valores exactos.

Enseguida vamos a seguir avanzando hacia la Estadística. Pero antes vamos a hacer un ejercicio y, tomando como punto de partida los resultados del ejercicio, nos detendremos un poco en algunos aspectos técnicos relacionados con el módulo `math` y el uso de funciones, aspectos que nos resultarán muy útiles más adelante.

### Ejercicio 1.

En este ejercicio vamos a utilizar las líneas de código que aparecen a continuación. Copia o teclea cada línea en el prompt (*¡practica las dos maneras!*), una por una, y ejecútala, pulsando Entrar tras copiar o teclear cada línea. Trata de adivinar el resultado de cada operación antes de ejecutar el código:

```
2 + 3
15 - 7
4 * 6
13 / 5
13 // 5
13 % 5
1 / 3 + 1 / 5
sqrt(25)
sqrt(26)
sin(pi)
sin(3.14)
```



## Prioridad de los operadores.

Fíjate en que en el ejercicio anterior Python ha interpretado el símbolo

```
1 / 3 + 1 / 5
```

como la operación

$$\frac{1}{3} + \frac{1}{5},$$

en lugar de darle otras interpretaciones posibles como, por ejemplo:

$$\frac{1}{\left(\frac{3+1}{5}\right)}.$$

Para hacer esa interpretación Python ha aplicado una serie de reglas, de lo que se conoce como **prioridad de operadores**, y que dicen en qué orden se realizan las operaciones, según el tipo de operador. No queremos entretenernos con esto ahora, pero podemos hacer un resumen básico diciendo que en operaciones como las que hemos visto:

1. Primero se calculan los valores de las funciones.
2. A continuación se evalúan productos y cocientes.
3. Finalmente se evalúan sumas y restas.
4. Dentro de cada uno de los pasos anteriores, siempre se evalúan las operaciones por orden de izquierda a derecha.

En caso de duda, o si necesitas alterar ese orden de las operaciones, siempre puedes (y a menudo, debes) usar paréntesis para despejar la posible ambigüedad. Por ejemplo, para distinguir entre las dos interpretaciones que hemos dado, puedes escribir:

```
(1 / 3) + (1 / 5)
```

o, por el contrario,

```
1 / ((3+1) / 5)
```

Un uso prudente de paréntesis y espacios en las operaciones es una marca característica del buen hacer, cuando se escribe código en un ordenador.

## Ejercicio 2.

Ejecuta esas dos operaciones para comprobar que obtienes los resultados esperados. Solución en la página 79. □

## Notación científica.

El resultado del cálculo de `sin(pi)` en el ejercicio anterior es `1.2246467991473532e-16`. La notación que se usa en la respuesta es la forma típica de traducir la notación científica a los lenguajes de ordenador y calculadoras. Ese símbolo representa al número:

$$1.2246467991473532 \cdot 10^{-16},$$

de manera que el número  $-16$ , que sigue a la letra  $e$  en esta representación, es el exponente de  $10$  (también llamado **orden de magnitud**), mientras que el número  $1.2246467991473532$  se denomina a veces **mantisa**. Puedes leer más sobre la notación científica en este artículo de la Wikipedia:

[http://es.wikipedia.org/wiki/Notaci%C3%B3n\\_cient%C3%ADfica](http://es.wikipedia.org/wiki/Notaci%C3%B3n_cient%C3%ADfica)

En cualquier caso, el exponente  $-16$  nos indica que se trata de un número extremadamente cercano a  $0$ . Recuerda que este resultado es una *aproximación* al valor exacto de  $\sin(\pi)$ , que es  $0$ . El propio símbolo `pi` de Python representa una aproximación y no debes confundirlo con el valor exacto de  $\pi$  en Matemáticas. Fíjate además en que si usas `3.14` como aproximación de  $\pi$  (como hemos hecho en el ejercicio), la respuesta, aunque pequeña, es todavía del orden de milésimas.

## Ayuda con Python.

Al trabajar con Python, como con cualquier otro lenguaje de programación, la aparición de errores es inevitable. Así que es conveniente saber lo que ocurre cuando le pedimos a Python una operación para la que no tiene respuesta. Por ejemplo, mira lo que sucede al ejecutar el siguiente código, que trata de dividir por cero:

```
In [11]: 3 / 0
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-11-2b706ee9dd8e> in <module>()
      1 3 / 0
ZeroDivisionError: division by zero

In [12]:
```

En este caso hemos mostrado todo el contenido de la consola desde la línea de entrada que produce el error, el consiguiente mensaje de error de Python y la siguiente línea de entrada de esa sesión. Como ves, Al hacerlo contesta con un mensaje de error que contiene una descripción más o menos detallada del tipo de error que se ha producido, en este caso `ZeroDivisionError`, y del punto concreto donde ha ocurrido el error (lo cual será muy útil cuando empecemos a escribir fragmentos de código más largos).

Veamos otros ejemplos de errores en el siguiente ejercicio.

### Ejercicio 3.

*Ejecuta consecutivamente estos comandos de Python, que producirán cada uno distintos tipos de errores, y fíjate en esos errores:*

```
log(-1)
```

```
4/*3
```

```
ln(7)
```



## Ayuda con Python.

¿Cómo se llaman las funciones de Python que calculan el arcotangente o el logaritmo en base 10? La respuesta a esta y a muchas otras preguntas está en Internet, y los buscadores son nuestros mejores aliados. Si hacemos la pregunta correcta, a menudo la primera respuesta de un buscador contendrá la información necesaria. Ten en cuenta que muchos de esos recursos están en inglés. En cualquier caso, existen algunos recursos sobre Python que es bueno conocer. El principal de ellos es la página oficial del lenguaje, situada en:

<https://www.python.org/>

y que contiene la documentación sobre los módulos oficiales con Python, tanto en la versión 2 como la 3. Por ejemplo, el módulo `math`, para la versión 3 de Python, está documentado en:

<https://docs.python.org/3/library/math.html>

### Ejercicio 4.

*Busca la respuesta a la pregunta que hemos dejado pendiente: ¿cómo se llaman las funciones de Python que calculan el arcotangente o el logaritmo en base 10?*



Otro recurso interesante son los foros (en inglés) de

[www.stackoverflow.com](http://www.stackoverflow.com)

No se trata de un foro más, donde cualquiera, con más o menos conocimientos puede opinar lo primero que se le ocurra. Es una comunidad con ciertas reglas y costumbres. Pero los usuarios que responden a las preguntas de esos foros son a menudo algunos de los mayores expertos del tema en concreto y las preguntas y respuestas son visibles para todos, pertenezcan o no a la comunidad y aparecen a menudo en los primeros lugares al usar un buscador. Precisamente por eso los mencionamos: si tu pregunta ya ha sido respondida en *stackoverflow*, seguramente la respuesta será muy detallada; abrumadoramente detallada en ocasiones. La comunidad de usuarios no está exenta de la inclinación natural de los humanos a pavonearse. Pero en cualquier caso, suelen ser discusiones interesantes de leer. Y tal vez con el tiempo tus preguntas lleguen a ser tan buenas que merezcan una discusión a fondo en *stackoverflow*. Aunque la comunidad de *stackoverflow* se centra en la programación, existen comunidades similares para otros temas. Por ejemplo *Biostars* para Genómica, *Cross Validated* para Estadística y Análisis de Datos, etc.

Finalmente, el propio Spyder nos puede proporcionar ayuda. Pero es mejor esperar a aprender un poco más sobre Python antes de usarla.

## 1.2. Más formas de importar funciones.

Recuerda que hemos visto que para usar funciones del módulo `math` tienes que usar una instrucción como:

```
from math import sqrt, sin, cos, tan, log, exp, pi, e
```

Es posible que te estés preguntando: "¿si voy a usar muchas funciones matemáticas tengo que importarlas escribiendo uno a uno el nombre de cada una de ellas?". Para evitar eso, que más adelante resultaría muy incómodo, Python proporciona varias formas alternativas de importar funciones desde un módulo.

En primer lugar, podemos decirle a Python que queremos importar *todas* las funciones de un módulo. Por ejemplo, para importar todas las funciones del módulo `math` usaríamos:

```
import math
```

El módulo `math` contiene una función llamada `sinh`, que sirve para calcular el seno hiperbólico de un número. No te preocupes si no sabes qué es y para qué sirve, es sólo un ejemplo; si te pica la curiosidad puedes ver su definición en la Wikipedia:

[https://es.wikipedia.org/wiki/Seno\\_hiperb%C3%B3lico](https://es.wikipedia.org/wiki/Seno_hiperb%C3%B3lico).

La función seno hiperbólico cumple:

$$\sinh(0) = 0$$

Así que, dado que supuestamente hemos importado todas las funciones del módulo `math`, deberíamos poder ejecutar este código y obtener 0 como respuesta (al menos aproximadamente). Sin embargo si ejecutas ese código verás que Python te espeta un mensaje de error inesperado:

```
In [10]: import math

In [11]: sinh(0)
-----
NameError                                 Traceback (most recent call last)
<ipython-input-11-f6b21eaa19a1> in <module>()
      1     sinh(0)

NameError: name 'sinh' is not defined
```

Como ves, Python dice que `name 'sinh' is not defined`. ¿Cómo es posible? Esto se debe a que la comodidad de importar todas las funciones del módulo `math` a la vez tiene un precio. Al usar `import math` Python en efecto ha importado todas las funciones, pero para saber de qué módulo proceden ha añadido el prefijo `math` seguido de un punto al nombre de cada una de esas funciones. Así que la forma correcta de usar la función es:

```
In [12]: math.sinh(0)
Out[12]: 0.0
```

que, como ves, ahora sí produce el resultado esperado.

## Importar módulos sin prefijo. Posibles conflictos de nombre.

Ahora es posible que pienses que, puestas así las cosas, no hemos ganado mucho importando todas las funciones de `math` a la vez. La supuesta comodidad queda en parte eclipsada por la necesidad de escribir ese prefijo `math` delante de cada aparición de una función del módulo.

Creemos que es conveniente que comprendas el problema que se plantea para los desarrolladores de Python, y el compromiso que han tenido que adoptar para evitar errores imprevisibles: pronto vamos a aprender a escribir nuestras propias funciones. Y estos prefijos son necesarios para evitar conflictos entre funciones de distintos módulos que tienen el mismo nombre.

En cualquier caso, cuando estamos muy seguros de lo que hacemos podemos usar una forma distinta de importación:

```
from math import *
```

Esta forma de importar te recordará a la primera que hemos visto, pero ahora el asterisco juega el papel de *comodín*, de manera que lo que estamos diciéndole a Python es que importe todas las funciones del módulo `math`, usando directamente los nombres de esas funciones, sin prefijos. Ahora puedes probar a ejecutar directamente:

```
sinh(0)
```

y comprobarás que no hay errores.

## Números complejos como ejemplo de conflicto de nombres.

Los conflictos de nombre a los que hemos aludido antes no son un fenómeno raro. Para que veas un ejemplo sencillo vamos a usar otro módulo de Python llamado `cmath` que contiene funciones para trabajar con números complejos. Podrías cargar todas las funciones de ese módulo como hemos hecho con las de `math`:

```
from cmath import *
```

Y ahora, supongamos que quieres volver a calcular la misma raíz cuadrada  $\sqrt{9}$  que vimos como primer ejemplo. ¿Qué sucede?:

```
In [13]: from cmath import *
```

```
In [14]: sqrt(9)
Out[14]: (3+0j)
```

La respuesta es un número complejo. No queremos entrar en detalles sobre los números complejos, porque no vamos a necesitarlos en el resto del curso. Estamos simplemente mostrando un ejemplo de cómo pueden aparecer los conflictos de nombres en cuanto se combinan varios módulos de Python. Si sabes algo sobre números complejos, lo único que debemos aclarar es que Python usa  $j$  para representar la unidad imaginaria (es decir,  $j^2 = -1$ ), la misma cantidad que a menudo se representa en los libros de matemáticas mediante  $i$  (la notación  $j$  es más frecuente en Física e Ingeniería).

¿Por qué ha ocurrido esto? Pues porque ambos módulos `math` y `cmath` tienen funciones llamadas `sqrt` pero que son *distintas*: la de `math` sirve para calcular la raíz cuadrada positiva de un número real positivo, mientras que la de `cmath` calcula una raíz cuadrada de cualquier número complejo, pero la respuesta es *siempre* un número complejo, independientemente de que el número de partida sea real o no. Y puesto que hemos importado `cmath` después de importar `math`, la función `sqrt` de `cmath` ha remplazado a la función `sqrt` de `math`. Y lo que es peor, en casos como este Python no nos avisa de que una función ha remplazado a otra (otros lenguajes de programación, como R, al menos lanzan una advertencia en situaciones similares).

### Ejercicio 5.

Vamos a comprobar el hecho de que el último módulo importado reemplaza a las anteriores funciones del mismo nombre. Vuelve a importar `math` usando el método del asterisco y repite el cálculo de `sqrt(9)`. ¿Qué sucede ahora? □

Como ilustra este ejemplo, la opción de importar las funciones de un módulo usando el asterisco es a menudo demasiado arriesgada y puede producir errores difíciles de diagnosticar cuando el código

en Python sea más complejo que los ejemplos básicos que estamos viendo. Especialmente cuando el código tiene más de un autor, que es la situación más habitual en el trabajo habitual. Por esa razón usar el asterisco para importar se considera una **mala práctica** al programar en Python. ¡No hagas! Y por si te lo has preguntado, ocurre algo análogo si usas:

```
from math import sqrt
```

y después

```
from cmath import sqrt
```

La segunda función importada reemplaza a la anterior. Tenemos que resignarnos, por tanto, a utilizar los prefijos de los módulos. Es decir que tenemos que hacer:

```
import math
```

Y ahora usar la función raíz cuadrada mediante:

```
math.sqrt(9)
```

Si queremos usar la raíz cuadrada de un número complejo hacemos:

```
import cmath
```

y ahora podemos calcular con:

```
cmath.sqrt(9)
```

Esto no afecta a la otra función `sqrt`, la de `math`, que sigue funcionando sin problemas. Veámoslo en una secuencia de comandos:

```
In [18]: import math
```

```
In [19]: math.sqrt(9)
```

```
Out[19]: 3.0
```

```
In [20]: import cmath
```

```
In [21]: cmath.sqrt(9)
```

```
Out[21]: (3+0j)
```

```
In [22]: math.sqrt(9)
```

```
Out[22]: 3.0
```

Como ves, la segunda llamada a `math.sqrt(9)` no se ve afectada por la función de `cmath`. Esta forma de trabajar elimina los conflictos de nombre entre módulos, pero puede resultar especialmente molesta con módulos de nombres largos. Por ejemplo, un poco más abajo aprenderemos a usar el módulo `matplotlib` para dibujar algunas gráficas. Sería bastante molesto tener que escribir el prefijo `matplotlib` cada vez que queremos usar una función de ese módulo. Para aliviar al menos parcialmente esa incomodidad Python nos permite usar un *alias*, normalmente una abreviatura, para importar un módulo. Por ejemplo, para importar `matplotlib` usaríamos:

```
import matplotlib as mp
```

y entonces en lugar de usar `matplotlib` como prefijo para las funciones de ese módulo basta con usar `mp`. Aunque `math` y `cmath` son nombres de módulo relativamente cortos, vamos a usar alias aún más cortos para que nos sirvan de ejemplo de cómo funciona esta idea. La secuencia anterior de comandos quedaría así:

```
In [23]: import math as m
In [24]: m.sqrt(9)
Out[24]: 3.0

In [25]: import cmath as c
In [26]: c.sqrt(9)
Out[26]: (3+0j)

In [27]: m.sqrt(9)
Out[27]: 3.0
```

Como ves, de esta forma el esfuerzo necesario para evitar conflictos es considerablemente menor.

### 1.3. Algunos detalles adicionales sobre Python. IPython.

Ya hemos dicho antes que Python es un lenguaje con una comunidad de usuarios muy amplia y que se usa de modos muy diversos, para tareas muy distintas. Eso se traduce en la existencia de multitud de herramientas distintas, cada una con sus pros y sus contras, adaptadas a esa gran diversidad del ecosistema Python. Y en particular, existen varias terminales posibles para trabajar con Python (en inglés se usa *shell* para referirse a la terminal). Nuestro contacto con la terminal hasta ahora se reduce a pensar en ella como un panel de Spyder que usamos para hablar con Python. Esta idea es en general correcta, pero queremos añadirle dos matices:

- En primer lugar, hay terminales que *viven* fuera de Spyder. Muchos programadores prefieren usar una terminal independiente, conectada con otras herramientas que les gustan más. Cuando tengas experiencia podrás decidir lo quéquieres hacer.
- Sin salir de Spyder, puedes utilizar más de una consola o más de un tipo de consola (se abrirían como pestañas del panel que contiene a la terminal).

La terminal de comandos que estamos usando es un tipo especial de terminal, llamada IPython. Este tipo de terminal añade algunas herramientas muy útiles al lenguaje Python básico. En este apartado vamos a empezar a ver algunas de ellas.

Si al llegar a este punto sientes una cierta confusión entre Python, IPython, Spyder, etc., no te preocupes. Es normal cuando te encuentras de golpe con muchos términos nuevos. Sigue adelante con las instrucciones que te proporcionamos y con la práctica todo irá quedando más claro.

#### Limpando la memoria de Python.

Al llegar a este punto hemos importado los módulos `math` y `cmath` de varias maneras y probablemente empieza a ser difícil seguirles el rastro. Y eso, al igual que ocurría con los conflictos de nombre, puede causarnos problemas en el resto de la sesión. A veces, al trabajar con IPython, te encontrarás en una situación como esta en la que quieras hacer *tabla rasa* y pedirle a Python que olvide todos los pasos previos para poder empezar a trabajar sin preocuparte de ese tipo de conflictos. Afortunadamente, existe un mecanismo para hacer esto. Basta con ejecutar este comando especial:

```
%reset
```

Al hacerlo, IPython nos pedirá que confirmemos esa decisión. Al fin y al cabo, estaremos borrando (casi) todo el trabajo previo de esa sesión. **Es muy importante entender esto.** Algunas sesiones de trabajo pueden contener cálculos muy valiosos, que tardan horas en ejecutarse, y en ese caso hacer un reset puede suponer perder todo ese trabajo. Asegúrate siempre de que realmente quieras hacer esto.

Una vez hechas las advertencias pertinentes, adelante. Y no te preocupes, a pesar de esas advertencias enseguida vamos a enseñarte un remedio para posibles despistes.

#### Ejercicio 6.

1. Ejecuta el comando

```
%reset
```

2. Prueba a ejecutar alguna de las operaciones que hemos hecho antes. Por ejemplo:

```
m.sqrt(9)
```

O también

```
math.sqrt(9)
```

O incluso

```
sqrt(9)
```

¿Qué sucede?

□

Una observación, antes de seguir adelante. Los comandos que empiezan por %, como `%reset` se denominan **comandos mágicos**. No son comandos de Python, sino de IPython y sólo sirven dentro de una terminal de IPython<sup>1</sup>. Más adelante volveremos sobre esto y entenderás mejor los matices de esa diferencia entre comandos mágicos y los comandos ordinarios de Python.

### El historial de comandos y el tabulador para completar código.

Al ejecutar el comando mágico `%reset` hemos borrado, como decíamos, casi toda la memoria de nuestra sesión de trabajo con Python. Pero aunque Python no recuerde nada, IPython sí recuerda algo que puede ser muy útil y valioso para nosotros: nuestro **historial de comandos**. Para verlo, sitúate en el prompt de la terminal de IPython y pulsa varias veces la tecla de la flecha hacia arriba en tu teclado. Al hacerlo verás como van desfilando, una tras otra, las últimas instrucciones que has tecleado en IPython, empezando por las más recientes. Y si pulsas la tecla de la flecha hacia abajo recorrerás esa lista en sentido inverso. En cualquier punto del recorrido puedes pararte y si lo deseas puedes hacer alguna modificación del comando que usaste anteriormente. Después puedes ejecutar el comando resultante (lo hayas modificado o no). Probemos esto:

#### Ejercicio 7.

Antes hemos usado Python para comprobar que:

$$\sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2} \approx 0.7071$$

Usa las flechas del teclado hasta localizar los comandos que usamos y modifícalos para comprobar estas otras identidades trigonométricas:

$$\begin{aligned} \sin\left(\frac{\pi}{3}\right) &= \frac{\sqrt{3}}{2}, & \cos\left(\frac{\pi}{3}\right) &= \frac{1}{2}, \\ \sin\left(\frac{\pi}{6}\right) &= \frac{1}{2}, & \cos\left(\frac{\pi}{6}\right) &= \frac{\sqrt{3}}{2}. \end{aligned}$$

□

Esta forma de navegar por el historial de comandos es muy cómoda cuando tienes que repetir un comando varias veces (quizá con pequeñas variaciones) o cuando se produce un error y debemos hacer correcciones. Además, Spyder nos proporciona una herramienta de seguridad adicional que complementa al historial de comandos de IPython y que nos puede sacar de más de un apuro. Si devuelves el panel de la terminal a su tamaño original (recuerda, con el ícono  de la barra superior de Spyder) verás que debajo de ese panel aparece una pestaña denominada **History Log**. Haz click con el ratón sobre ella y verás que un nuevo panel ocupa el espacio de la terminal (a la que puedes volver con la pestaña denominada **IPython console**). El panel **History Log** muestra nuestro *historial de comandos* de esta y otras sesiones previas (las fechas y horas de comienzo de cada sesión aparecen como comentarios). La siguiente figura muestra el panel History Log de Spyder (la versión para Mac en este caso) en el que puedes ver algunos de los últimos comandos que han ejecutado en esa sesión (terminando con un `%reset`).

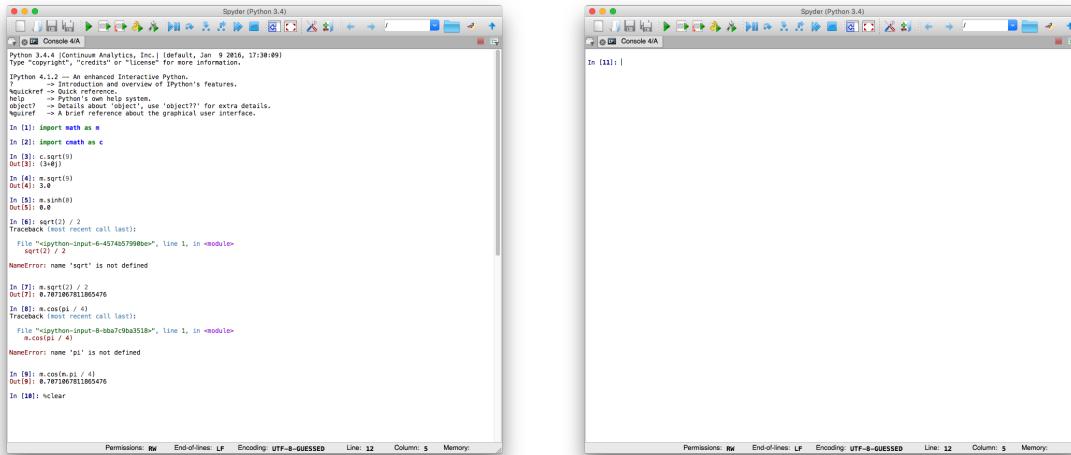
<sup>1</sup> Y que los expertos nos perdonen esta simplificación. Ya veremos que en realidad sí se pueden usar fuera de la terminal.

The screenshot shows the Spyder IDE's 'History log' window. It displays a list of Python commands entered in the IPython console, including arithmetic operations like division and multiplication, imports from the math module (sin, cos, tan, log, exp, pi), and imports from cmath. At the bottom of the list, the command '%reset' is visible.

Una posibilidad, si quieres poner a salvo tu trabajo, es copiar el contenido de esta ventana (al menos la parte que te interesa proteger) y pegarlo en un documento del editor de texto (*Bloc de Notas* o similar). Puedes guardar ese documento como copia de respaldo de tu trabajo. Pero pronto, antes del final de este tutorial, te mostraremos una manera mejor de hacer esto. Así que lo mejor es que pienses en el *History Log* como un último recurso a utilizar para situaciones imprevistas. Aparte de eso yo lo uso para hacer copia/pega de comandos, combinándolo con el historial de comandos.

### Limpieza visual de la consola de IPython.

Regresemos a la terminal. Hay otro comando mágico que tal vez quieras usar a veces al trabajar con IPython. Se trata del comando `%clear`. Su efecto se entiende mejor comparando las siguientes dos imágenes. En la de la izquierda estamos en medio de una sesión de trabajo con IPython, en la que hemos introducido diversos comandos, hemos cometido errores, etc. Y estamos justo a punto de ejecutar `%clear`. A la derecha se muestra el resultado después de ejecutarlo:



Como ves, `%clear` limpia el contenido del panel con la terminal de IPython. A diferencia de `%reset`, la memoria de Python no se ve afectada. Y tampoco se elimina el historial de comandos, el efecto es puramente visual. Pero a menudo nos resulta cómodo “despejar” la pantalla para poder trabajar con más claridad.

## 2. Variables y listas en Python.

En la sección previa hemos usado Python como una calculadora. Pero, para ir más allá, tenemos que disponer de **estructuras de datos**. Ese término describe, en Computación, las herramientas que nos permiten almacenar y procesar información. Las estructuras de datos más básicas de Python

son las **variables** y las **listas**. En este y en próximos tutoriales nos iremos encontrando con otras estructuras de datos: conjuntos, diccionarios, tuplas, matrices, dataframes, entre otras.

## 2.1. Variables en Python.

Te recomendamos que antes de seguir adelante hagas una limpieza completa en la terminal de IPython. Puedes usar los comandos mágicos `%reset` y `%clear` o, si lo prefieres, puedes simplemente cerrar Spyder y volver a abrirlo (en Anaconda no hace falta que cierres el Launcher, basta con Spyder).

Una variable en Python es un símbolo o nombre que usamos para referirnos a un objeto. Por ejemplo, ejecuta este código:

```
a = 2
```

Aparentemente no ha sucedido nada. En la consola de IPython no hay respuesta: no aparece inmediatamente una línea de salida (que empezaría por `Out`). Pero a partir de ese momento Python ha **asignado** el valor 2 al símbolo `a`. Así que si, por ejemplo, ejecutas

```
a + 1
```

ahora sí que verás una línea de salida con el resultado que imaginas. La secuencia completa es esta:

```
In [1]: a = 2  
In [2]: a + 1  
Out[2]: 3
```

Podemos crear una variable con una instrucción muy sencilla, como `a = 2`, pero también como resultado de efectuar en el lado derecho una operación mucho más complicada. Por ejemplo, después de importar el módulo `math` con el alias `m` vamos a usar una variable `V` para calcular el volumen de una esfera de radio  $r = 10\text{cm}$ . Recuerda que el volumen viene dado por:

$$V = \frac{4}{3}\pi r^3.$$

Así que usamos estos comandos:

```
In [4]: import math as m  
In [5]: V = (4 / 3) * m.pi * 10**3
```

Eso está muy bien, pero ¿cuánto vale `V`? Hay dos formas de ver ese valor. En IPython lo más rápido es escribir el nombre de la variable y ejecutarlo como una instrucción:

```
In [6]: V  
Out[6]: 4188.790204786391
```

Pero también podemos usar la función `print` así:

```
In [7]: print(V)  
4188.790204786391
```

En este caso no hay diferencia y eso puede llevarte a pensar que el primer método nos ahorra trabajo. Y en efecto así es, siempre que busquemos una respuesta rápida, en casos sencillos y mientras estamos trabajando en IPython. Pero a medida que avancemos por los tutoriales pronto tendrás ocasión de aprender más sobre `print` y verás que en muchos casos es una opción mejor y en otros, sencillamente, es la única forma de llegar a la información que queremos. Un comentario más, antes de que se nos olvide: no hemos necesitado importar `print` desde ningún módulo porque es una de las funciones básicas de Python, que están disponibles *siempre* en cualquier sesión de trabajo.

**Advertencia sobre Python 2:** ya dijimos en el Tutorial-00 que avisaríamos al lector cuando nos encontráramos con diferencias importantes entre Python 2 y Python 3. Pues bien, el funcionamiento de la función `print` en Python 2 es distinto del que estamos mostrando aquí. Por ejemplo, en Python 2 la función no usa paréntesis. Así que es correcto escribir:

```
print 2 + 3
```

mientras que en Python 3 esto produciría un mensaje de error. Si vas a usar Python 2 es **imprescindible** que aprendas bien esas diferencias en el uso de `print`. Aquí no vamos a profundizar más allá de esta advertencia, porque como ya hemos advertido sólo vamos a usar Python 3.

### Asignaciones.

Las instrucciones de Python como

```
a = 2
```

o como

```
V = (4 / 3) * m.pi * 10**3
```

que tienen la estructura:

```
variable = expresion
```

se llaman **asignaciones** y decimos que se asigna el resultado de la expresión de la derecha a la variable que aparece a la izquierda. Lo más importante que hay que recordar sobre las asignaciones es que el valor que se asigna reemplaza a cualquier valor que hubiera almacenado en la variable previamente. Así, por ejemplo, si hacemos

```
a = 2
```

y después

```
a = 3
```

el valor 2 que inicialmente estaba asignado a la variable `a` se pierde. Si no se tiene en cuenta esto es fácil cometer errores al sobrescribir valores.

### Ejercicio 8.

¿Cuánto valen las variables `a`, `b` y `c` al ejecutar estos comandos uno tras otro? Haz una tabla con tres columnas tituladas `a`, `b` y `c` y anota el valor de las variables en cada paso.

```
a = 2
b = 3
c = a + b
a = b * c
b = (c - a)**2
c = a * b
```

Como has podido ver, al ejecutar esas asignaciones Python no produce ningún valor como resultado.  
¿Se te ocurre alguna forma de comprobar los resultados que has escrito en la tabla? □

### Copiando bloques de código a IPython.

Hasta ahora se supone que para trabajar en la consola de IPython has ido o copiando y pegando uno a uno o tecleando los comandos de Python que te sugerímos. Pero cuando aparezcan fragmentos de código más largos en estos tutoriales en algún momento esa operación de copiar y pegar una a una las líneas de código resultará molesta. Hay una forma más rápida de trabajar que te permite copiar bloques enteros de código. Para practicarlo, selecciona estas cuatro líneas de código (asegúrate de que las tienes seleccionadas todas)

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

cópialas y pégalas en la consola de IPython. Deberías ver algo como esto:

```
In [1]: a = 2
....: b = 3
....: c = a + b
....: print(a, b, c)
```

con el cursor parpadeando tras el último paréntesis de la cuarta fila. **¡Esto es importante para lo que sigue!** Si el cursor no está situado en esa posición asegúrate de usar las flechas del teclado para llevarlo hasta ahí.

Si el pegado no ha ido bien y el texto que has obtenido no es lo que esperabas es mejor que esperes un poco más, hasta que aprendamos a usar el editor de código de Spyder. ¡La culpa, en estos casos, no es de Spyder! Depende, entre otras cosas, del programa que uses para leer el pdf de este tutorial. En esta figura puedes ver un ejemplo de un pegado que ha ido mal, porque los saltos de líneas no se han conservado al pegar:

```
In [12]:
In [13]: a=2
....: b=3 c=a+b print(a, b, c)
```

Puedes desplazarte dentro de ese bloque de texto con las flechas de cursor y pulsar Enter en las posiciones en las que quieras que aparezcan saltos de línea. Pero cuando nos vemos obligados a hacer esto, arreglando *a mano* el código, las ventajas del copia/pega por bloques empiezan a difuminarse.

En cualquier caso, una vez que hayas conseguido que el bloque de código aparezca correctamente y suponiendo que el cursor está situado en la última posición de la última línea de ese bloque, pulsa *Enter*. Aparecerá una línea en blanco más porque IPython te está dando la oportunidad de añadir más instrucciones.

```
In [1]: a = 2
....: b = 3
....: c = a + b
....: print(a, b, c)
....:
```

Pero como no es el caso y no queremos añadir nada, pulsamos *Enter* una vez más y, ahora sí, IPython le envía a Python ese bloque de instrucciones una tras otra, Python las ejecuta y el resultado en pantalla es:

```
In [1]: a = 2
....: b = 3
....: c = a + b
....: print(a, b, c)
....:
2 3 5
```

```
In [2]:
```

Además de aprender a trabajar con bloques de código, hemos aprovechado este fragmento de código para ilustrar otra forma de usar la función `print` para mostrar a la vez los valores de varias variables. Es posible que a la vista de esto quieras volver sobre el ejercicio 8 (pág. 15).

### Nombres de las variables y palabras reservadas.

Aunque hasta ahora hemos usado letras como nombres de las variables, puedes utilizar nombres más descriptivos. Y muchas veces es una buena idea hacerlo. Por ejemplo, puede que hace una semana hayas escrito estas instrucciones para resolver un problema:

```
a = 2  
b = 3  
c = a / b
```

Pero si las vuelves a ver, pasada una semana, es muy probable que no recuerdes qué era lo que estabas tratando de conseguir al hacer esto. En cambio, al ver estas instrucciones:

```
espacio = 2  
tiempo = 3  
velocidad = espacio / tiempo
```

es mucho más fácil reconocer el objetivo que persiguen. A lo largo de los tutoriales del curso vamos a insistir muchas veces en la necesidad de que el código esté bien *organizado*, y esté bien *documentado*. Un primer paso en esa dirección es tratar de elegir nombres descriptivos para las variables. En Python las reglas para los nombres de variables son muy flexibles: esencialmente, que empiecen por una letra y no contengan espacios ni caracteres especiales, como ?, +, paréntesis, etcétera. Tampoco puedes usar la ñ ni letras acentuadas<sup>2</sup>. Pero puedes usar un guión bajo \_ como parte del nombre y a veces se hace para hacer más legibles las variables. Por ejemplo:

```
temp_final
```

para representar la temperatura final de un proceso. Pero cuidado con los excesos. Es cierto que puedes usar nombres de variables arbitrariamente largos. Pero si usas como nombre:

```
Estavariablealmacenaelresultado delaoperaciontaninteresante queacabamosdehacer
```

tu trabajo resultará ilegible. Como siempre, se necesita un equilibrio, y con la práctica encontrarás el tuyo (consejo zen gratuito del tutorial de hoy). Para empezar, es una buena idea combinar mayúsculas y minúsculas en los nombres de las variables. Volviendo al ejemplo de la temperatura final de un proceso, el nombre **temperaturafinalproceso** es menos legible y más largo que **tempFinal**. Es mucho más fácil, por otra parte, que te equivoques tecleando el primero de esos nombres. Otro aspecto que debes tener en cuenta al elegir nombres para tus variables es que no debes usar como nombres aquellos símbolos que forman parte del propio lenguaje Python. Por ejemplo, hemos visto que en Python las palabras **sqrt**, **import**, **math**, **from**, **print** se usan como parte de las instrucciones del lenguaje. Es habitual referirse a estas palabras como **palabras reservadas** del lenguaje. En algunos lenguajes de programación esas palabras están de hecho reservadas y si tratas de usarlas como nombre de variable se producirá un error. Pero Python es más tolerante y te permite usar algunas de esas palabras como nombre de variable. Esa flexibilidad puede ser conveniente para programadores muy expertos, pero tú **¡no lo hagas!** No suele ser una buena idea y produce errores que pueden ser muy difíciles de detectar. Al programar en español es fácil buscar nombres alternativos para las variables y según cuál sea tu campo de trabajo es posible que te cueste imaginar una situación en la que usarías **import** como nombre de variable. Pero insistimos, procura tener cuidado con la elección de los nombres de variable para hacerlos útiles y evitar conflictos.

## Variables de tipo cadena de caracteres.

En el Capítulo 1 del libro hemos hablado de variables cualitativas y cuantitativas. Estas últimas toman siempre valores numéricos, y las variables de Python sirven, desde luego, para almacenar esa clase de valores. Pero, como iremos viendo en sucesivos tutoriales, también se pueden utilizar variables de Python para guardar valores de variables cualitativas (factores), y otros tipos de objetos que iremos conociendo a lo largo del curso. De momento, para que veas a qué nos referimos, recordaremos el Ejemplo 1.1.1 del libro (pág. 6), en el que teníamos una variable cualitativa ordenada que representa el pronóstico de un paciente que ingresa en un hospital. Prueba a ejecutar este código:

```
pronostico = "leve"
```

simplemente para que, de momento, veas que:

<sup>2</sup>En general, también se desaconseja usar cualquiera de esos caracteres en los nombres de ficheros, carpetas, etc. Para algunos sistemas operativos no supone ningún problema, pero en otros casos puede crearte auténticos quebraderos de cabeza. Es el precio que pagamos por usar una tecnología pensada para el juego de caracteres del idioma inglés.

- Python no protesta. El valor "leve" es un valor perfectamente aceptable.
- En Python los valores que representan palabras o frases se denominan **cadenas alfanuméricas** (**cadenas de caracteres** (en inglés *character strings*, a menudo abreviado simplemente a *strings*)). Las cadenas de caracteres se escriben siempre entre comillas. Puedes usar comillas dobles, como en "leve" o simples, como en 'leve'.

En los próximos tutoriales tendremos ocasión de extendernos sobre la relación entre los factores y los valores alfanuméricos de Python. Pero la utilidad de las variables de tipo alfanumérico va mucho más allá. Las usaremos pronto para añadir títulos y otras etiquetas a los gráficos o tablas, para añadir mensajes informativos a los cálculos que hagamos, etc. Y en un sentido más amplio, el procesamiento de cadenas alfanuméricas es el punto de partida de campos de trabajo como la Genómica (el genoma se representa mediante cadenas de caracteres que contienen la secuencia de bases del ADN presente en los cromosomas) o el procesamiento del lenguaje natural (el que hablamos las personas), sin el cual no dispondríamos de herramientas como buscadores avanzados de Internet, o el reconocimiento de voz, etc. Hay, por lo tanto, un mundo por descubrir cuando se trabaja con este tipo de variables, a las que apenas nos hemos asomado. Al final de este curso no serás ni mucho menos un experto, pero habrás aprendido los rudimentos del trabajo con ese tipo de variables que te permitirán pasar a textos más avanzados si lo deseas. De momento, un aperitivo.

### Ejercicio 9.

Ejecuta estas instrucciones (recuerda que puedes copiarlas como un bloque):

```
mensaje1 = "¡Hola, "
usuario = "Alicia"
mensaje2 = "!, ¿cómo estás?"
print(mensaje1 + usuario + mensaje2)
```

¿Qué se obtiene como salida? Como puedes comprobar, la operación suma + en el caso de cadenas de caracteres da como resultado la concatenación de esas cadenas. Prueba a cambiar la segunda línea por

```
usuario = "Luis"
```

y ejecuta de nuevo el código. Las operaciones como las de este ejemplo son frecuentes en los sistemas que producen mensajes personalizados para el usuario y, más en general, son una herramienta básica para crear textos mediante programas.



### Tipos de variables.

En la Sección 1.1.1 del libro (pág.. 5) hemos discutido las diferencias entre los tipos de variables más comunes en Estadística: variables cuantitativas, que pueden ser discretas o continuas, y variables cualitativas, también llamadas factores. Python también clasifica a sus variables en tipos. Y aunque la correspondencia entre los tipos de variables en Estadística y en Python no se puede establecer automáticamente, es necesario conocer los tipos de Python para poder elegir el tipo más adecuado para representar en el código los valores de una variable estadística.

En Python las variables de tipo **int** (del inglés *integer*, entero) sirven para almacenar números enteros (por tanto positivos, 0 o negativos). Cuando hacemos una asignación como:

```
a = 12
```

Python examina el valor que estamos usando y automáticamente asigna el tipo **int** a la variable **a**. Para comprobar el tipo de una variable disponemos de la función **type**, como puedes ver:

```
In [1]: a = 12
In [2]: type(a)
Out[3]: int
```

¿Qué sucede cuando la variable representa un número decimal? Compruébalo:

**Ejercicio 10.** Asigna el valor 7.32 a la variable **b** y usa la función **type** para descubrir de qué tipo es la variable resultante.



Como has podido comprobar en este ejercicio, Python usa el tipo `float` para la representación decimal de los números. Ese tipo de variables y los valores que almacenan se suelen llamar en español de *coma flotante* (en inglés es *floating point*, de ahí el nombre `float`). Los valores en notación científica que hemos visto en la página 6 también son de tipo `float`.

Las variables cuantitativas discretas se representan en Python de manera natural mediante variables de tipo `int`, mientras que las variables cuantitativas continuas se pueden representar con variables de tipo `float`, siempre teniendo en cuenta que se trata de aproximaciones a los números reales. Una variable de tipo `float` sólo puede almacenar unas cuantas cifras decimales de un número como  $\frac{1}{3}$ , que en realidad tiene una representación decimal periódica  $0.33333\dots$  (con infinitas cifras). Es importante recordar esto para evitar confusiones al interpretar los resultados del código Python. ¿Qué ocurre con los factores, las variables cualitativas? Cuando un factor toma sólo  $n$  valores (recuerda, decimos que el factor tiene  $n$  niveles) podemos codificar esos niveles mediante los números del 1 al  $n$  y por tanto usar una variable de tipo `int`. Aunque eso es sin duda posible, en otras ocasiones preferiremos usar variables de tipo cadena de caracteres, para poder asignar nombres más informativos a los niveles del factor. Es mucho más sencillo entender el código si vemos una asignación como:

```
pronostico = "leve"
```

que si viéramos en su lugar:

```
pronostico = 2
```

¿De qué tipo es la variable creada al asignarle como valor una cadena de caracteres?

```
In [1]: pronostico = "leve"  
In [2]: type(pronostico)  
Out[2]: str
```

Como ves, se trata de variables de tipo `str` (del inglés *string*, que a su vez es una abreviatura de *character string*, cadena de caracteres).

Hay mucho más que decir sobre la representación y el uso de los factores en Python. En futuros tutoriales volveremos sobre este tema y sobre los tipos de variables de Python en general.

Antes de cerrar este apartado, queremos mencionar que existen numerosas funciones de Python para convertir valores de un tipo de variable a otro tipo. Esas conversiones pueden ser muy útiles, pero no hay que perder de vista que en ocasiones la conversión lleva aparejada una pérdida de información. Algunas de esas funciones se llaman exactamente igual que el tipo de destino al que queremos convertir un valor. Por ejemplo, para convertir de tipo `int` (entero) a `float` (coma flotante) tenemos la función `float`:

```
In [1]: float(7)  
Out[1]: 7.0
```

Fíjate en que el resultado muestra claramente que `float` da como resultado la representación decimal del número (la representación en coma flotante, para ser precisos). ¿Y el camino inverso? Usamos la función `int`:

```
In [1]: int(4.67)  
Out[1]: 4
```

Aquí tienes un ejemplo claro de pérdida de información. La conversión de decimal a flotante implica eliminar las cifras decimales después del punto con la consiguiente pérdida de precisión. Mira otro ejemplo:

```
In [2]: int(-4.67)  
Out[2]: -4
```

Como ves, la función `int` simplemente elimina las cifras decimales, con independencia de que el número sea positivo o negativo. Este resultado no coincide, por tanto, con la idea de *redondear al entero más cercano*. Afortunadamente, para eso disponemos de la función `round`, que hace eso:

```
In [3]: round(4.67)
```

```
Out[3]: 5
```

```
In [4]: round(-4.67)
```

```
Out[4]: -5
```

De hecho, `round` hace más. Podemos usar un segundo argumento para pedirle a `round` que produzca un resultado con una cierta cantidad de cifras decimales, como en este ejemplo:

```
In [5]: round(-4.67, 1)
```

```
Out[5]: -4.7
```

¡Ten en cuenta que se trata de cifras decimales y no significativas! Más adelante en este tutorial vamos a aprender a obtener un resultado con una cierta cantidad de cifras significativas, usando la función `print` que ya conocemos. Y a su debido tiempo hablaremos de otras conversiones entre tipos de variables.

## División en Python 2.

Una advertencia: en Python 2, una división escrita como `a/b` se interpreta como división entera si ambos operandos `a` y `b` son enteros. Pero basta con que uno de ellos sea un número en coma flotante (de tipo `float`) para que el resultado de `a/b` en Python 2 sea un `float`. Muchas veces se ha dicho (y yo también lo creo) que esa ambigüedad en la interpretación es la principal causa de errores en muchos programas escritos en Python 2. Afortunadamente, Python 3 es más simple.

## 2.2. Listas en Python

En Estadística, lo habitual es trabajar con colecciones o *muestras* de datos. Y para almacenar esas colecciones de datos, la estructura más básica de Python son las listas. En este apartado vamos a empezar nuestro trabajo con ellos, aunque a lo largo del curso aún tendremos ocasión de aprender bastante más sobre el manejo de las listas de Python.

Para empezar vamos a trabajar con listas que contienen una colección de números, que pueden ser las edades de los alumnos de una clase:

```
22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19
```

En Python la lista que corresponde a esas edades se construye mediante un comando como este:

```
edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19]
```

En realidad, en esa expresión hemos hecho dos cosas:

1. Hemos *creado* la lista con los datos, en la parte derecha de la expresión. Los datos están separados por comas, y rodeados por corchetes.
2. Una vez creada la lista, la hemos *asignado* a la variable `edades`. Hasta ahora sólo habíamos usado variables para identificar un único valor (un número o una cadena alfanumérica). Pero una variable puede usarse para identificar una lista o, como veremos más adelante, estructuras de datos mucho más complejas.

Una observación más: Python permite mezclar en una misma lista valores de tipos distintos. Por ejemplo números y cadenas alfanuméricas, o listas de listas. Por ejemplo, prueba a ejecutar:

```
alumno = ["Alicia", "López", 17, [7, 8.3, 7.2, 6.8, 8.3]]
```

En esta orden hemos creado una lista con los datos de una alumna: nombre, apellido, edad y una lista de sus notas en cinco asignaturas. Además, hemos asignado esa lista a la variable `alumno`. Si las cosas van bien, no esperes ninguna respuesta: como ya hemos dicho, Python no muestra el resultado de las asignaciones. Para comprobar que Python nos ha entendido ejecuta:

```
print(alumno)
```

Vamos a hacer algunas operaciones con la lista de edades de alumnos que hemos creado antes. Imagínate que, como sucede a menudo, después de haber creado nuestra lista de edades, desde la administración nos avisán de que hay cinco alumnos nuevos, recién matriculados, y debemos incorporar sus edades, que son

a nuestra lista de datos. Naturalmente, podríamos empezar de nuevo, creando una lista completa desde cero. Pero es preferible reutilizar la lista `edades` que ya habíamos creado. Vamos a ver esto como primer ejemplo, para empezar a aprender cómo se manipulan listas en Python. Una forma de añadir los nuevos datos es empezar creando una segunda lista que los contiene:

```
edades2 = [22, 18, 20, 21, 20]
```

Y a continuación *concatenamos* las dos listas. La concatenación se representa en Python mediante el símbolo de suma `+`, como ya hemos visto para las cadenas de caracteres:

```
edades = edades + edades2
```

Puesto que hemos terminado con una asignación, Python no muestra ninguna salida y eso puede hacer que te resulte difícil seguir lo que ha ocurrido. En la siguiente captura de una sesión de trabajo en IPython hemos añadido algunos pasos adicionales para tratar de ayudarte a analizar lo que sucede:

```
In [12]: edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19]
In [13]: edades2 = [22, 18, 20, 21, 20]
In [14]: print(edades + edades2)
[22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19, 22, 18, 20, 21, 20]
In [15]: edades = edades + edades2
In [16]: print(edades)
[22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19, 22, 18, 20, 21, 20]
```

Vamos a analizar paso a paso lo que ha ocurrido:

- En las dos primeras líneas creamos la lista original de edades y la lista con las edades adicionales que queremos incorporar.
- En la línea de entrada In [14] se muestra (usando `print`) lo que ocurre al concatenar ambas listas con `+`. Como ves el resultado es la lista que esperábamos, con los elementos de `edades2` situados tras los de `edades`. Hemos añadido esta línea para que veas el resultado de esa operación. Pero puesto que no hemos asignado ese resultado a ninguna variable, el resultado *se ha perdido*.
- Por eso en la línea In [15] repetimos la operación, pero esta vez asignamos el resultado a la variable `edades`. Al ser una asignación no hay mensaje de salida. Es muy importante que comprendas que al hacer esto hemos hecho una **resignación**. Y por tanto el contenido original de la lista `edades` se ha perdido. En este caso lo hemos hecho a conciencia, porque queríamos *actualizar* el valor de esa variable después de concatenar la segunda lista de edades. Pero en otros casos podríamos estar interesados en conservar la lista original. En esos casos esta reasignación habría sido un grave error.
- Y para comprobar lo que decíamos en el paso anterior, en la línea In [16] hemos usado `print` para que veas el contenido de la lista `edades` tras ejecutar el código anterior.

Vamos a hacer un ejercicio para insistir en estas ideas, porque entender bien lo que hemos hecho aquí es esencial para nuestro trabajo futuro.

**Ejercicio 11.** 1. Para empezar, vamos a descubrir una de las tareas que `print` hace por nosotros. Vuelve a ejecutar las tres primeras instrucciones del grupo anterior, pero ahora sin usar `print` en la tercera. Es decir, ejecuta:

```
edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19]
edades2 = [22, 18, 20, 21, 20]
print(edades + edades2)
```

¿Qué ha sucedido?

2. Viendo que hemos “sumado listas”, tal vez te preguntes: ¿se pueden restar listas? Haz la prueba. Ejecuta este bloque de instrucciones y mira lo que sucede:

```
lista1 = [1, 2, 3, 4, 5, 6]
lista2 = [4, 5, 6]
print(lista1 - lista2)
```

□

### Selección de elementos dentro de una lista.

En un ejemplo anterior hemos creado la lista `alumno` con este comando:

```
alumno = ["Alicia", "López", 17, [7, 8.3, 7.2, 6.8, 8.3]]
```

Supongamos ahora que queremos usar el nombre de esta alumna en alguna operación, por ejemplo para imprimir una lista de notas. Para eso tenemos que *acceder* a la información que está almacenada dentro de cada una de las posiciones de la lista `alumno`. La forma de acceder al nombre es esta

```
alumno[0]
```

En la terminal de IPython se obtiene:

```
In [19]: alumno[0]
Out[19]: 'Alicia'
```

Y el apellido se obtiene con:

```
In [20]: alumno[1]
Out[20]: 'López'
```

Lo más importante que tienes que observar es que **en Python las posiciones dentro de una lista se empiezan a contar desde 0**. Esto contrasta con lo que sucede en otros lenguajes de programación, en los que se empieza a contar desde 1. Las dos opciones tienen sus ventajas y sus inconvenientes. Para los programadores experimentados de Python resulta natural trabajar así, pero para los que se inician en el lenguaje (o para quienes tenemos que cambiar a menudo de un lenguaje a otro) esta característica de Python suele suponer una fuente de errores muy común. No podemos hacer mucho más, aparte de pedirte paciencia y animarte a prestar atención a esto. Vamos a avanzar un paso en nuestra capacidad de seleccionar elementos dentro de una lista. Recuerda que hemos creado una lista de edades que en su última versión es:

```
print(edades)
[22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19, 22, 18, 20, 21, 20]
```

Supongamos que ahora necesitamos extraer de aquí una lista con las edades de los cinco primeros alumnos de lista. En Python podemos hacerlo así:

```
edades[0:5]
```

Al ejecutar este comando se obtiene:

```
edades[0:5]
Out[28]: [22, 21, 18, 19, 17]
```

Fíjate en que hemos escrito entre corchetes `[0:5]`. En Python hay que acostumbrarse a leer esto así: las posiciones de la 0 a la 5, **sin incluir la 5**. De nuevo, sirve la advertencia: olvidarse de que se excluye la última posición es un error típico.

Por ejemplo, para obtener las posiciones de la sexta a la décima de la lista usaríamos:

```
In [29]: edades[5:10]
Out[29]: [21, 18, 20, 17, 18]
```

Y la lógica pythonesca detrás de `[5:10]` es esta:

- Empezamos en 5 porque al contar desde 0 el 5 en realidad ocupa la sexta posición.
- Terminamos en 10 porque al contar desde 0 el 10 ocupa la undécima posición, *pero en realidad este último elemento no se incluye*, con lo que realmente la última posición incluida es la décima.

Sí, lo sabemos: las primeras veces esto resulta bastante embrollado. Y como decíamos, salvo que llegues a ser un programador experimentado de Python, es muy posible que tengas que pensar con atención cada operación similar a estas.

El primer ejemplo que hemos visto, el de las cinco primeras posiciones de la lista representa una situación que se da muy a menudo: queremos las  $n$  primeras posiciones de una lista, siendo  $n$  un número cualquiera. En ese caso podemos ahorrarnos el cero inicial dentro del corchete. Es decir, que por ejemplo las cinco primeras posiciones se obtienen también con:

```
In [30]: edades[:5]
Out[30]: [22, 21, 18, 19, 17]
```

Otro paso más. Supongamos que queremos seleccionar cinco elementos de la lista, empezando desde el primero (posición 0, recuerda) pero saltando de tres en tres. Haríamos:

```
In [31]: edades[0:15:3]
Out[31]: [22, 19, 18, 18, 20]
```

Puedes comprobar que hemos conseguido lo que queríamos. El 3 final de `[0:15:3]` es la forma en la que le indicamos a Python que queremos avanzar de tres en tres posiciones. Y hemos llegado a 15 porque queremos 5 elementos y  $3 \cdot 5 = 15$ . Pero piénsalo despacio: la posición 15 no se incluye. ¿No se pierde entonces uno de los cinco elementos? No, porque empezamos a contar desde 0. A riesgo de ser pesados insistimos en esto porque al principio es muy posible que te cueste acostumbrarte al funcionamiento de Python.

### Ejercicio 12.

1. ¿Qué sucede si en lugar de `edades[0:15:3]` usas `edades[0:14:3]`?
2. ¿Y si usas `edades[0:16:3]`?

□

En otras ocasiones es posible que queramos obtener la última posición o, por ejemplo, las últimas cinco posiciones de la lista. La última posición se obtiene con:

```
In [32]: edades[-1]
Out[32]: 20
```

La penúltima es:

```
In [33]: edades[-2]
Out[33]: 21
```

A la vista de estas operaciones, ¿qué harías para obtener las últimas cinco posiciones de la lista? No te hemos propuesto esta tarea como un ejercicio porque no es fácil que aciertes. Es posible que pienses en hacer:

```
In [34]: edades[-5:-1]
Out[34]: [22, 18, 20, 21]
```

Pero como ves eso no ha funcionado: de la misma forma que al escribir `[2:7]` el 7 no se incluye, al escribir `[-5:-1]`. Es posible que entonces pienses en usar `[-5:0]`. Pero eso tampoco funcionará, y de hecho Python te dará como respuesta una lista vacía:

```
In [35]: edades[-5:0]
Out[35]: []
```

A Python no le gusta especialmente que mezclemos índices negativos con índices no negativos (como el 0).

¿Cuál es entonces la solución? Hace un rato vimos que `edades[:5]` nos proporcionaba los cinco primeros elementos. Pues una construcción similar nos proporciona los cinco últimos:

```
In [36]: edades[-5:]
Out[36]: [22, 18, 20, 21, 20]
```

Dejamos este apartado con un ejercicio para que practiques estas construcciones:

### Ejercicio 13.

1. ¿Qué se obtiene al usar `edades[-1:-6:-1]`? Te aconsejo que mires la lista original para no despistarte.
2. ¿Y al usar `edades[-1:-6:-2]`?
3. ¿Y si tratas de hacer `edades[-1:-6:-2]`, qué ocurre? ¿Por qué?
4. Finalmente, ¿qué se obtiene con `edades[:]`? ¿Y con `edades[::-1]`? Presta atención a este último truco porque es especialmente útil.

□

### Objetos de tipo range (recorrido).

En el apartado anterior, al tratar de seleccionar las posiciones de una lista nos hemos encontrado con un tipo especial de objeto que es muy frecuente en operación. Se trata de listas de números enteros (números que representan las posiciones), ya sea consecutivos o separados por una cantidad fija (técnicamente, hablamos de progresiones aritméticas). Es decir, una lista como la que forman los números del 1 al 10:

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

o una lista como esta:

```
7, 10, 13, 16, 19, 22
```

en la que la diferencia entre cada dos números consecutivos de la lista es 3, o también una lista como esta otra:

```
21, 19, 17, 15, 13
```

en la que los números disminuyen en lugar de aumentar.

Como tendremos ocasión de ver a lo largo del curso, este tipo de listas son el armazón sobre el que se van trabando muchos de los algoritmos que construiremos. Por esa razón Python, como muchos otros lenguajes, le da un tratamiento especial a estas listas. En Python, de hecho, se trata a estas listas como un tipo especial de objetos, objetos de tipo `range` y existe una función especial para fabricarlos: la función se llama igual que los objetos, `range`. Por ejemplo la primera de las listas que hemos visto, los números del 1 al 10, se corresponde con el objeto que creamos así:

```
range(1, 11)
```

¿Por qué 11? Por la misma razón que hemos visto al seleccionar elementos de una lista: en Python no se incluye el último elemento. Si ejecutas esa función en la terminal de IPython ocurrirá algo como esto:

```
range(1, 11)
Out[1]: range(1, 11)
```

Y de hecho, puesto que no hemos asignado ese objeto a una variable, es como si no hubiéramos hecho nada. Volvamos a intentarlo asignándolo a una variable. La asignación no produce salida, como sabes, así que usaremos `print` a continuación:

```
In [2]: uno_a_diez = range(1, 11)
```

```
In [3]: print(uno_a_diez)
range(1, 11)
```

Bueno, eso confirma que la asignación ha funcionado, pero no nos dice mucho más. La razón por la que no estamos viendo los números del 1 al 10 es porque, insistimos, `uno_a_diez` no es una lista, sino un objeto de tipo `range`. Estas diferencias entre tipos de objetos son a veces sutiles y sin duda son una de las dificultades con las que se enfrentan los recién llegados a Python. Pero tienen su sentido: sirven para que los programas que escribamos sean más eficientes. En cualquier caso, tenemos a nuestra disposición una función que convierte los objetos de tipo `range` en objetos tipo `lista`. La función se llama sencillamente `list`. Así que si haces:

```
In [4]: print(list(uno_a_diez))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

ahora sí, podemos comprobar que ese objeto describe la lista de enteros que queríamos. De la misma forma, la lista de números

```
7, 10, 13, 16, 19, 22
```

se obtiene con

```
range(7, 23, 3)
```

#### Ejercicio 14.

1. Compruébalo usando `list` y `print`.
2. ¿Qué ocurre al usar `range(7, 22, 3)`?
3. ¿Y si usas `range(7, 24, 3)`?
4. Usa la función `range` para representar la lista

```
21, 19, 17, 15, 13
```

y comprueba el resultado usando `list` y `print`.

5. ¿Podemos usar incrementos fraccionarios? Prueba a usar

```
range(1, 11, 0.5)
```

□

Más adelante veremos la respuesta al problema que se plantea en el último apartado de este ejercicio. Pero antes, en la próxima sección vamos a empezar a usar las listas para hacer Estadística.

#### Operaciones sobre una lista en conjunto: `sum` y `len`.

Volvamos a la lista de edades. Recuerda que era:

```
In [15]: print(edades)
[22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19, 22, 18, 20, 21, 20]
```

¿Cómo calculamos la edad media a partir de esta lista? Necesitamos, para empezar, una forma de sumar los elementos de la lista. Afortunadamente esto es muy fácil: en Python existe una función `sum` que sirve precisamente para sumar una lista de números:

```
In [16]: sum(edades)
Out[16]: 486
```

Para calcular la media sólo nos falta dividir por el número de elementos de la lista. ¿Cuántos son? En este ejemplo son suficientemente pocos como para que tengas aún la tentación de contarlos. Pero pronto encontraremos ejemplos con cientos, miles o incluso millones de datos. Y está claro que contar no es una opción. Pero de nuevo Python nos proporciona justo la función que necesitamos, que se llama `len` (del inglés *length*, longitud) y que calcula la longitud de una lista:

```
In [18]: len(edades)
Out[18]: 25
```

Con eso tenemos todos los ingredientes para calcular la edad media:

```
In [19]: mediaEdad = sum(edades) / len(edades)

In [20]: print(mediaEdad)
19.44
```

Las funciones `len` y `sum` son sólo algunas de las funciones que existen en Python para trabajar con una lista completa. Antes de terminar el tutorial veremos unas cuantas funciones más del mismo tipo.

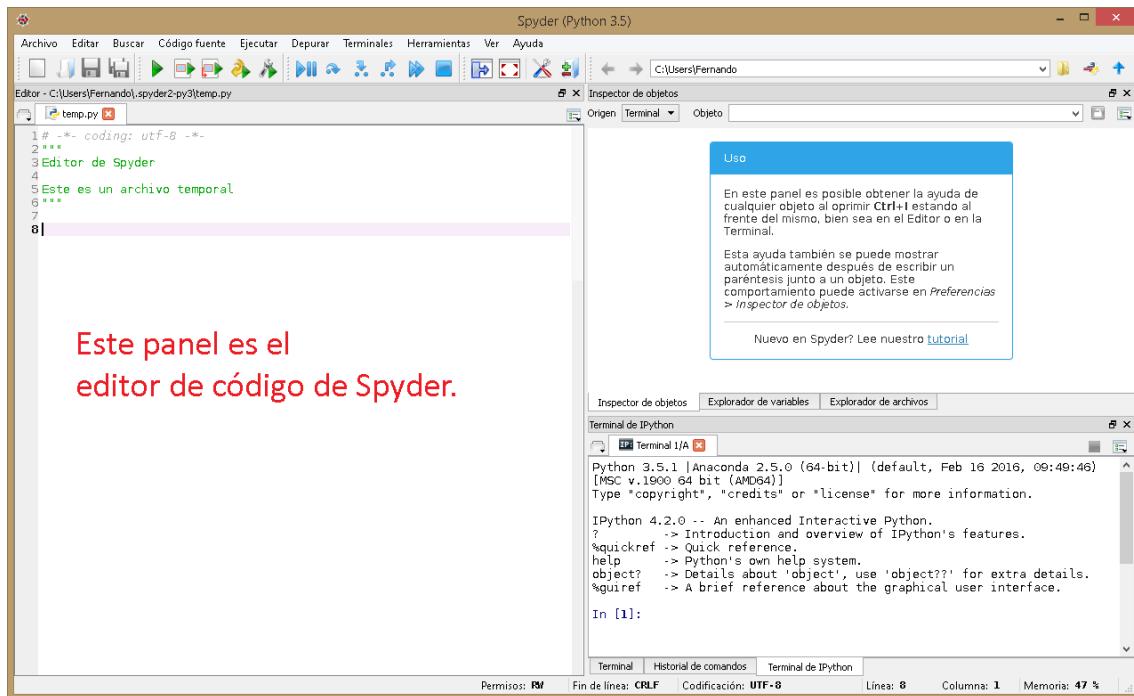
El siguiente paso natural podría ser calcular la varianza de `edades` (para empezar calcularemos la poblacional, aunque eso no es relevante). Y aquí tropezamos con una dificultad que nos va a obligar a profundizar en nuestros conocimientos de Python. Para calcular la varianza uno de los pasos (ver la Sección 2.3.2 del libro, pág. 35) consiste en elevar al cuadrado cada elemento de una lista de valores. Y eso es precisamente lo que aún no hemos aprendido a hacer: aplicar una operación *a cada uno de los elementos de una lista por turno*. Es una de las tareas más comunes en programación y en las próximas secciones vamos a empezar a aprender cómo se hace.

### 3. El editor de código y ficheros de código Python.

Pero antes tenemos que dar otro paso esencial para poder avanzar como programadores Python. Hasta ahora nuestro trabajo ha consistido básicamente en ejecutar líneas de código en la terminal, una tras otra, y observar los resultados que Python produce como respuesta. Esa forma de trabajar es suficiente para muchas cosas y seguramente la seguirás usando en el futuro (por ejemplo, es una forma natural de trabajar al explorar un nuevo conjunto de datos). Además hemos visto que podemos pegar en la terminal bloques formados por varias líneas de código Python. Pero también vimos que ese método copia/pega no estaba exento de inconvenientes. De hecho, tiene bastantes limitaciones si nos queremos plantear cualquier trabajo avanzado con Python.

Porque a menudo necesitaremos trabajar con conjuntos de instrucciones más sofisticado, auténticos *programas* escritos en Python por nosotros mismos o por otros programadores. Y para poder hacer eso vamos a usar otra componente de Spyder, el panel que llamaremos *Editor de código*. Para asegurarnos de que nuestro trabajo anterior no interfiere con lo que vamos a hacer ahora, lo mejor es que cierres y vuelvas a abrir Spyder antes de continuar. Y en cualquier caso, recuerda que si has usado el icono para maximizar el panel de la terminal, puedes usarlo de nuevo para hacer visibles el resto de paneles de Spyder, incluido el *Editor de código*.

Al abrir Spyder, por defecto, el *Editor de código* ocupa la parte izquierda de la ventana de Spyder como se muestra en esta figura (en la versión para Windows de Spyder).



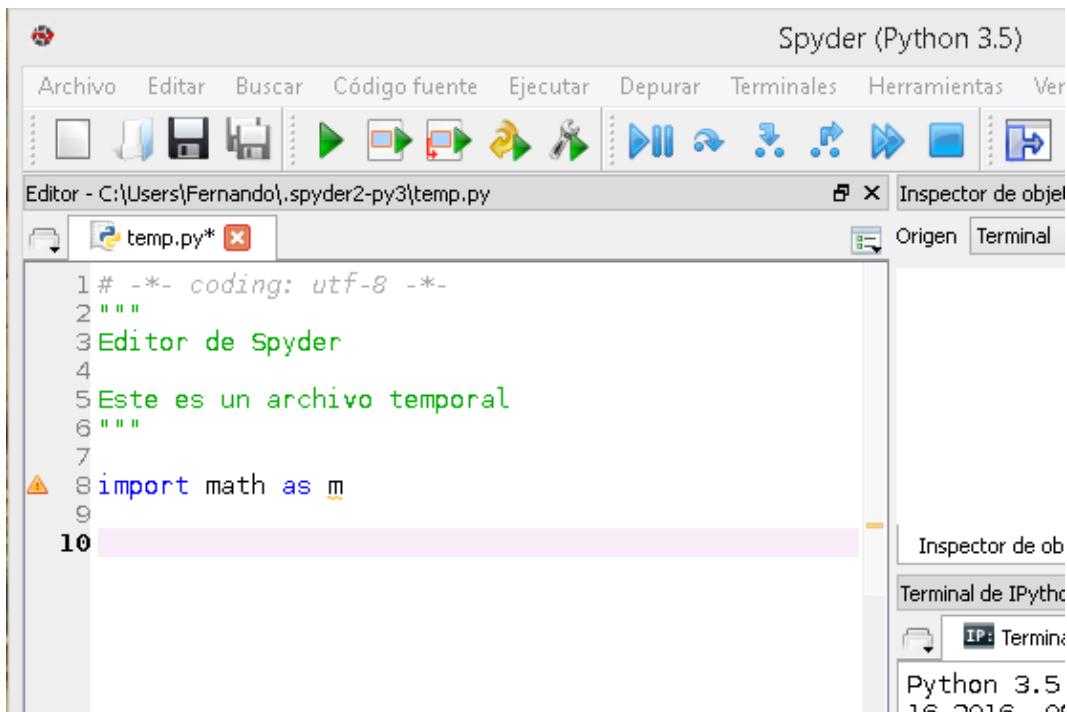
Este panel es el editor de código de Spyder.

Verás que el panel del editor contiene en la parte superior algunas líneas de texto (que pueden ser distintas de las que aparecen aquí). No te preocupes por ellas, porque no van a interferir en nuestro trabajo. Este panel se comporta en muchos sentidos como un editor de texto, al estilo del *Bloc de Notas* de Windows. Pero como irás viendo, es un editor de texto especialmente preparado para el trabajo con Python.

Para empezar a trabajar con el Editor de Código, haz click en ese panel y asegúrate de que el cursor esté situado por debajo de las líneas del principio. A continuación escribe esta línea de código:

```
import math as m
```

En este momento deberías ver algo así en el Editor de código:



Algunas observaciones:

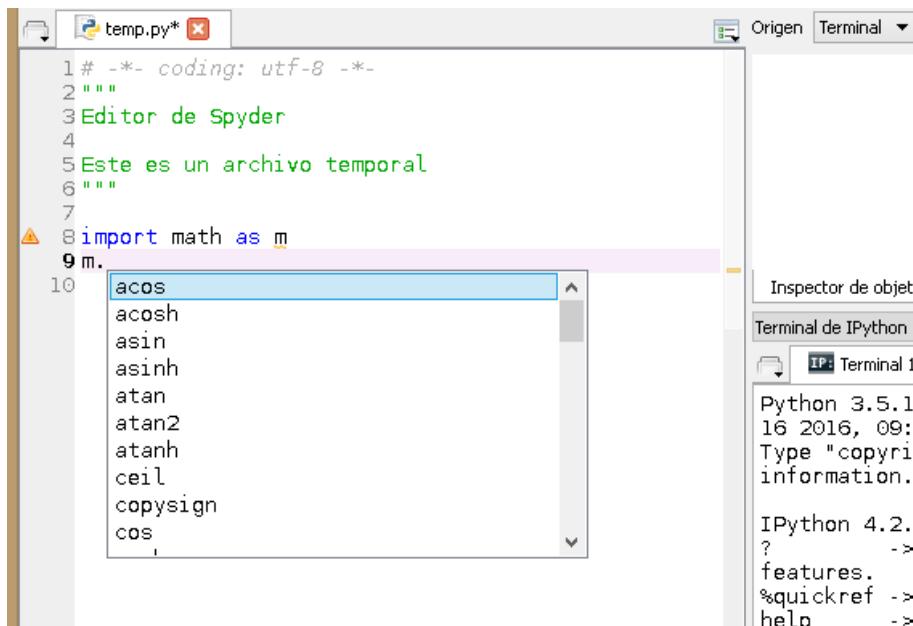
- Por el momento no te preocupes por el triángulo amarillo de advertencia que ha aparecido a la izquierda de esa línea. Esos mensajes de Spyder te serán muy útiles más adelante, pero por ahora lo mejor es ignorarlo (salvo que sea de color rojo, en cuyo caso probablemente significa que has copiado mal el código).

- Si te fijas verás que las palabras `import` y `as` aparecen en color azul, mientras que `math` y `m` son de color negro. La diferencia es que las dos primeras son palabras reservadas del lenguaje Python, mientras que `math` y `m` son identificadores, como los nombres de variables. Este *código de colores* es el segundo ejemplo que vemos de la forma en la que el Editor de Texto nos ayuda al escribir código en Python (el primer ejemplo fue el triángulo amarillo de advertencia).

Hemos dicho que este panel de Spyder es parecido al *Bloc de Notas*. Pero es un bloc de notas que incluye estas características especiales, como el reconocimiento de sintaxis para Python. Eso significa que se usan colores, tipos de letra, etc. para destacar la estructura del programa y ayudarnos así en nuestro trabajo. Además, como iremos viendo, el Editor de Código vigila que nuestro programa cumpla unas normas básicas de sintaxis Python, ahorrándonos los errores más evidentes (de esa forma Spyder se asegura de que cuando cometamos errores, al menos sean errores interesantes...) Todavía hay más. Vamos a añadir una línea de código más a ese panel, justo debajo de la anterior. Queremos añadir una línea para calcular la raíz cuadrada de 2. Ya sabes que para eso podemos usar:

```
m.sqrt(2)
```

Cuando empiezas a teclear esta línea, justo después de teclear el punto que contiene esa línea te encontrarás con que aparece esto:



¿Qué está ocurriendo?

- Spyder ha visto la primera línea y sabe que vamos a usar `m` como alias del módulo `math`.
- Por lo tanto, al escribir `m.` Spyder sabe que nos vamos a referir a una función de ese módulo, y nos muestra una lista con todas las funciones que contiene (lo cuál puede ser muy útil si sólo recuerdas aproximadamente el nombre de la función).

En cualquier caso no te preocupes, sigue escribiendo esa línea de código hasta el final. Cuando escribas el primer paréntesis verás que Spyder añade el segundo paréntesis (para que no nos olvidemos, como sucede a menudo con expresiones complejas con paréntesis anidados) y retrocede una posición para que podamos seguir escribiendo cómodamente dentro de ese par de paréntesis. Añade el 2 hasta obtener:

The screenshot shows the Spyder Python IDE interface. The title bar reads "Spyder (Python)". The menu bar includes "Archivo", "Editar", "Buscar", "Código fuente", "Ejecutar", "Depurar", "Terminales", and "Herramientas". Below the menu is a toolbar with various icons. The main window shows an editor titled "temp.py\*" containing the following code:

```
1 # -*- coding: utf-8 -*-
2 """
3 Editor de Spyder
4
5 Este es un archivo temporal
6 """
7
8 import math as m
9 m.sqrt(2)
10
```

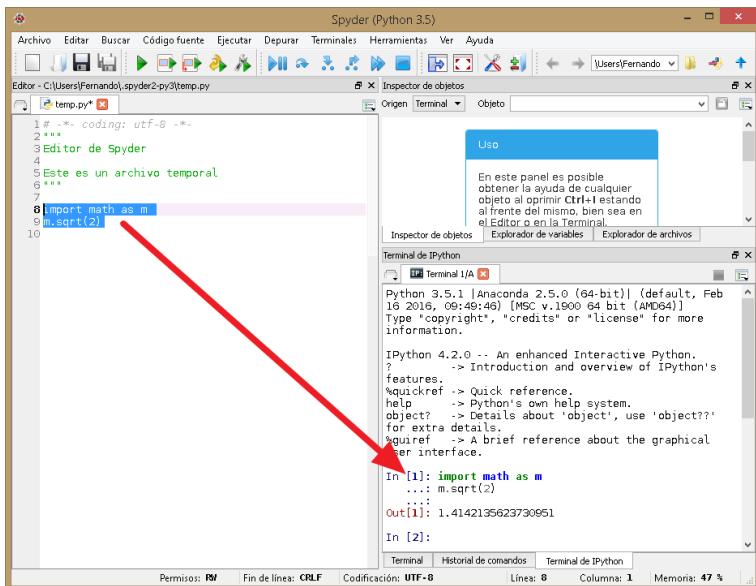
The code is color-coded: green for strings and comments, blue for the "import" keyword, and red for the "m" alias. Lines 8 and 9 are highlighted with a light purple background, indicating they are selected.

Un detalle más: al situar el cursor a la izquierda del segundo paréntesis verás que Spyder *ilumina o resalta* en color verde los dos paréntesis de la pareja, de nuevo como ayuda visual para que sea fácil identificar que paréntesis izquierdo corresponde a cada paréntesis derecho.

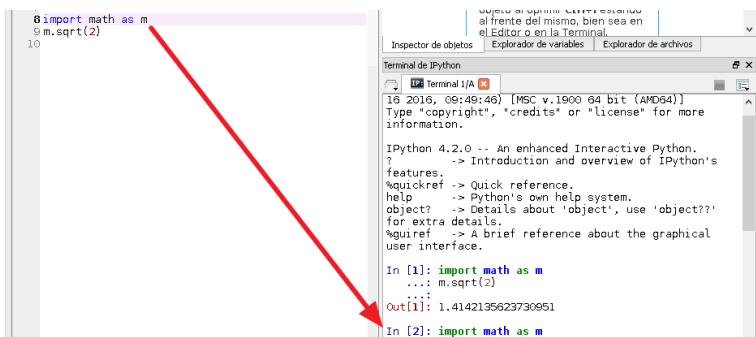
Ahora empieza la parte divertida. Usando el teclado o el ratón, selecciona las dos líneas de código que has tecleado, como aparecen en esta figura:

The screenshot shows the Spyder Python IDE interface with the same code as before. The lines "8 import math as m" and "9 m.sqrt(2)" are now highlighted with a light purple background, indicating they have been selected.

Y ahora, en el menú *Ejecutar (o Run)* de Spyder (los nombres de opciones varían ligeramente entre Windows, Mac y Linux), selecciona la opción *Ejecutar la Selección (Run selection)* o simplemente, pulsa la tecla de función F9. Al hacerlo fíjate en lo que sucede en la terminal de IPython:



En efecto, Python ha ejecutado esas líneas de código y se muestra el resultado. De hecho no hace falta ejecutar las dos líneas a la vez. Si sitúas de nuevo el cursor en la primera línea, sin seleccionarla por completo (basta con que el cursor esté en esa línea) y pulsas F9 (o usas el menú):



Como puedes comprobar sólo se ejecuta esa línea. Haz lo mismo con la segunda línea para ver cómo funciona.

Esta conexión entre esos dos paneles de Spyder, el Editor de Textos por un lado y la Terminal de Ipython por otro, nos va a servir para trabajar con mucha comodidad: normalmente escribiremos el código de nuestros programas en el Editor de Textos. De esa forma aprovechamos toda la ayuda que nos presta, y de la que ya hemos empezado a ver algunas muestras. Y cada vez que queramos ver cómo funciona una parte de nuestro código, usamos esa conexión para ejecutarlo. Aunque al principio te puede costar añadir este nuevo ingrediente a la mezcla, pronto te acostumbrarás y verás que resulta una forma sencilla de trabajar.

Un experimento más. Añade una nueva línea al *Editor de Código*, con esta asignación:

```
a = 2
```

y ejecuta esa línea usando F9. Como era de esperar, la línea se ejecuta en la terminal (y no hay línea de salida). Ahora haz clic con el ratón **en la terminal** y ejecuta:

```
print(a + 1)
```

Como resultado obtendrás 3, claro. No hay ninguna sorpresa en esto, y sólo queremos reforzar la idea de que el *Editor de Código* y la *Terminal* están realmente conectados a través del mecanismo que hemos visto. Por supuesto, esta última línea de código (con `print`) que hemos escrito en la *Terminal* no aparece por ningún sitio en el *Editor de Código*. Más adelante veremos la utilidad de esto.

### 3.1. Ficheros de comandos Python.

El *Editor de Código* nos va a servir para escribir programas Python. Esos programas se almacenan en ficheros de texto simple, con extensión .py para identificarlos fácilmente y para que nuestro ordenador sepa qué aplicación utilizar para abrir esos ficheros. En principio, puedes abrir y modificar

uno de esos ficheros con cualquier editor de texto, como el **Bloc de Notas** de Windows. Pero, como ya hemos visto, es mejor usar el editor de código que incorpora Spyder, porque así disponemos de reconocimiento de la sintaxis de Python y de otras ayudas para el trabajo de programación.

Esos ficheros de código Python se pueden intercambiar fácilmente con otros usuarios, permitiéndonos así compartir programas útiles. Para empezar a practicar el manejo de esos ficheros incluimos aquí como adjunto un fichero de código Python especialmente simple, que se limita a calcular y mostrar el volumen y área de una esfera cuyo radio se define mediante el valor de la variable **r** que aparece en la segunda línea del código.

### Tut02-py-VolAreaEsfera.py

y cuyo contenido se muestra a continuación:

```
import math as m

r = 2

V = (4 / 3) * m.pi * r**3

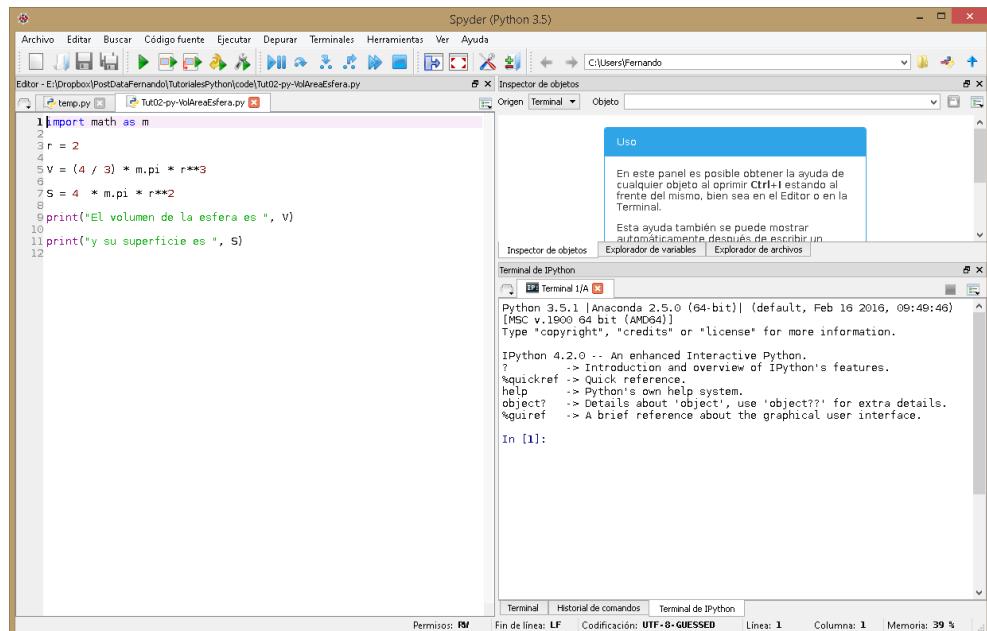
S = 4 * m.pi * r**2

print("El volumen de la esfera es ", V)

print("y su superficie es ", S)
```

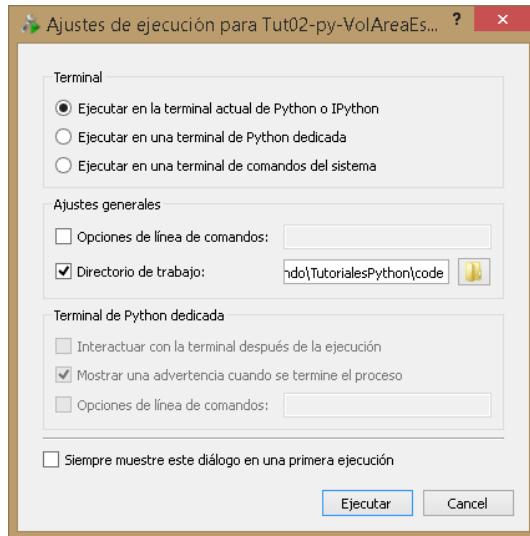
Como ves, hemos dejado líneas en blanco entre cada dos líneas de código. No es necesario hacer esto y en cualquier caso al ejecutar el programa Python va a ignorar esas líneas vacías. Pero a menudo puede ser útil como recurso visual, para hacer más legible el programa. Ya veremos más recomendaciones de estilo, necesarias para convertirnos en programadores con buenas prácticas del oficio.

Vamos a abrir este fichero en el *Editor de Código* de Spyder. Empieza por guardar ese fichero (recuerda usar el botón derecho del ratón para descargarlo del pdf) en la carpeta **codigo** de tu *Directorio de Trabajo*. Esta es la carpeta en la que vamos a guardar todos los ficheros de código Python del curso y algunos no funcionarán si no están en esa carpeta, así que es **importante** que te acostumbres a esa organización desde el principio. Es recomendable, una vez más, que cierres y vuelvas a abrir Spyder antes de continuar, para empezar el trabajo en un entorno limpio. Una vez hecho eso, usa el menú *Archivo* y la opción *Abrir* de Spyder. En la ventana que se abre (y que depende de tu sistema, Windows, Mac o Linux) navega hasta tu directorio de trabajo y selecciona el fichero **Tut02-py-VolAreaEsfera.py** que contiene nuestro programa. Spyder lo abrirá en una pestaña del *Editor de código*, y verás algo como lo que se muestra en esta figura:

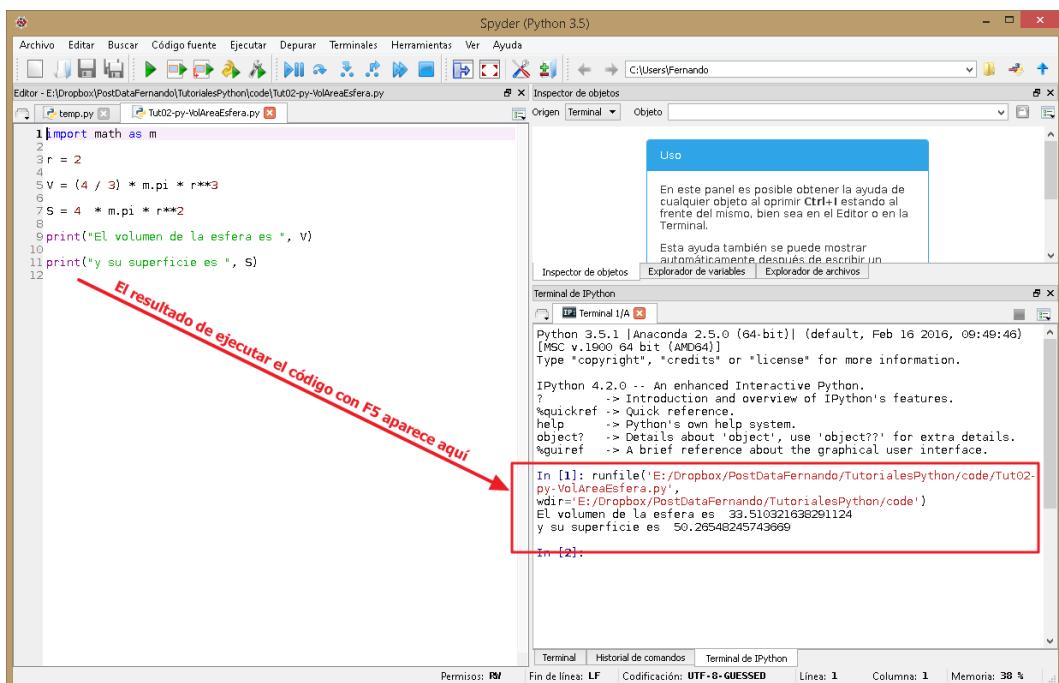


Como puedes ver, Spyder está aplicando reconocimiento de sintaxis Python al fichero y usando colores para ayudarnos a identificar los componentes de ese programa. Una observación antes de seguir: la primera pestaña normalmente contendrá el fichero temporal `temp.py` que Spyder crea siempre al comenzar para almacenar el código que aún no hemos guardado en un fichero. Puedes hacer click en ella para revisar su contenido. Es posible que esa primera pestaña contenga aún código de nuestra sesión de trabajo previa. No te preocunes, no interferirá con lo que vamos a hacer. Vuelve a la pestaña que contiene el programa `Tut02-py-VolAreaEsfera.py`.

Ya hemos visto antes que puedes ir ejecutando una por una las líneas de código que aparecen en el *Editor* (recuerda, basta con usar F9 con el cursor situado en la línea que quieras ejecutar). Pero en el caso de un programa completo como este, podemos ejecutar todo el código de una vez pulsando F5. Al ser la primera vez que ejecutamos código, aparecerá un cuadro de diálogo como este:



Podemos aceptar todas las opciones por defecto, así que haz click en *Ejecutar* (*Run* si tu versión de Spyder está en inglés). En la *Terminal* aparecerá el resultado de la ejecución, como se ve en esta figura:



Verás una clara diferencia con lo que sucedía cuando ejecutábamos el código línea a línea. Ahora, para ejecutar nuestro código Python ha usado una función llamada `runfile` (con dos opciones, que corresponden al nombre del fichero de código y al nombre del directorio de trabajo). Y en la terminal no vemos las líneas de código que componen el programa, sino solamente la salida que produce ese código (mediante `print`).

Para ver más clara la diferencia, vamos a añadir una línea más al código en la que calculamos la relación entre el volumen y el área de la esfera. Es decir, añadimos esta línea de código justo al

final del fichero:

V / S

Después de añadir esa línea vuelve a pulsar F5 (al hacerlo, Spyder graba automáticamente el fichero de código con los cambios que hemos hecho). Como verás, en la terminal aparece de nuevo el resultado de ejecutar el código, pero no hay ningún cambio visible como respuesta a esa nueva línea.

```
9 print("El volumen de la esfera es ", v)
10
11 print("y su superficie es ", s)
12
13 V / S
```

```
Python 3.5.1 |Anaconda 2.5.0 (64-bit)| (default, Feb 16 2016, 09:49:46)
[MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?   -> Details about 'object'; use 'object??' for extra details.
%gui?     -> A brief introduction to the graphical user interface.

In [1]: runfile('E:/Dropbox/PostDataFernando/TutorialesPython/code/Tut02-py-VolAreaEsfera.py', wdir='E:/Dropbox/PostDataFernando/TutorialesPython/code')
El volumen de la esfera es 33.510321638291124
y su superficie es 50.26548245743669

In [2]: runfile('E:/Dropbox/PostDataFernando/TutorialesPython/code/Tut02-py-VolAreaEsfera.py', wdir='E:/Dropbox/PostDataFernando/TutorialesPython/code')
El volumen de la esfera es 33.510321638291124
y su superficie es 50.26548245743669

In [3]:
```

Si ahora sitúas el cursor en esa última línea del *Editor de Código* y pulsas F9 (para ejecutar sólo esa línea) el resultado que verás en la terminal es:

```
In [3]: V / S
Out [3]: 0.6666666666666666
```

La razón por la que sucede esto es que, tras ejecutar el programa, Python sabe ahora cuáles son los valores de V y S. Al ejecutar esa línea con F9, usando lo que llamaremos **modo interactivo**, Python se comporta como una calculadora que muestra el resultado de esa operación.

Vamos a insistir en la idea. Cambia esa línea para que sea:

```
RazonVS = V / S
```

y vuelve a usar F5. Como antes, no hay efecto *visible* de esa última línea de código. Pero el código se ha ejecutado. Si te sitúas en la terminal, escribes **RazonVS** y pulsas *Enter* para ejecutarlo, verás que Python sabe cuál es el valor, porque lo ha calculado al ejecutar el programa.

```
import math as m
r = 2
S = 4 * m.pi * r**2
print("El volumen de la esfera es ", v)
print("y su superficie es ", s)
RazonVS = V / S
```

```
Python 3.5.1 |Anaconda 2.5.0 (64-bit)| (default, Feb 16 2016, 09:49:46)
[MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?   -> Details about 'object'; use 'object??' for extra details.
%gui?     -> A brief introduction to the graphical user interface.

In [1]: runfile('E:/Dropbox/PostDataFernando/TutorialesPython/code/Tut02-py-VolAreaEsfera.py', wdir='E:/Dropbox/PostDataFernando/TutorialesPython/code')
El volumen de la esfera es 33.510321638291124
y su superficie es 50.26548245743669

In [2]: runfile('E:/Dropbox/PostDataFernando/TutorialesPython/code/Tut02-py-VolAreaEsfera.py', wdir='E:/Dropbox/PostDataFernando/TutorialesPython/code')
El volumen de la esfera es 33.510321638291124
y su superficie es 50.26548245743669

In [3]: V / S
Out[3]: 0.6666666666666666

In [4]: runfile('E:/Dropbox/PostDataFernando/TutorialesPython/code/Tut02-py-VolAreaEsfera.py', wdir='E:/Dropbox/PostDataFernando/TutorialesPython/code')
El volumen de la esfera es 33.510321638291124
y su superficie es 50.26548245743669

In [5]: RazonVS
Out[5]: 0.6666666666666666

In [6]:
```

Como sucede con todas las novedades que estamos aprendiendo en este tutorial, al principio te puede costar acostumbrarte al vínculo entre la *Terminal* y el *Editor de Código* de Spyder. Pero con la práctica y a medida que avancemos por estos tutoriales, todo irá quedando más claro y llegarás a dominar estas herramientas. A partir de ahora verás que nuestra atención se desplaza cada vez más hacia el *Editor de Código*, como lugar natural en el que escribir con comodidad el código Python.

### Un comentario sobre otros editores de texto.

Hemos repetido varias veces que muchos programadores prefieren trabajar con otras herramientas, en lugar de usar un entorno de desarrollo integrado como Spyder. ¿Por qué? Una de las posibles razones es que a menudo un programador utiliza más de un lenguaje de programación en su trabajo. Es común usar una combinación de Python con R, o con C++, JavaScript, etc. Y los entornos de desarrollo como Spyder a menudo son especialistas en un lenguaje, pero no se llevan igual de bien con los otros. Por esa razón a veces los programadores prefieren trabajar directamente con editores de texto, similares al *Bloc de Notas* pero más potentes, que reconocen la sintaxis de muchos lenguajes de programación. En el Tutorial00 hemos mencionado varios de ellos. Por ejemplo: Notepad++ en Windows, TextWrangler en Mac OS X, gedit en Linux. En cualquier caso los ficheros de texto simple son una parte esencial del trabajo en Computación y un buen editor de texto simple es un aliado importante para cualquier programador o científico. Así que, aunque sigas usando Spyder para programar en Python, te recomendamos que te familiarices con uno de ellos.

## 4. Bucles for.

Como hemos dicho, una de las operaciones básicas en computación consiste en aplicar una operación a cada uno de los elementos de una lista. Nos hemos encontrado ya con una situación en la que queríamos hacer esto, al tratar de calcular la varianza poblacional de la lista `edades`. Recuerda que la lista es:

```
edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19,
18, 22, 20, 19, 22, 18, 20, 21, 20]
```

y que ya hemos calculado su media, que es  $\frac{486}{25} = 19.44$ . El cálculo se hacía con

```
mediaEdad = sum(edades) / len(edades)
```

La definición de la varianza poblacional de una lista  $x_1, x_2, \dots, x_n$  es:

$$Var(x) = \frac{(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \dots + (x_n - \bar{x})^2}{n}$$

En la Sección ?? (pág. ??) del Tutorial01 hemos hecho esto con la hoja de cálculo. Allí vimos una descripción de esa fórmula en forma de receta o **algoritmo** para realizar el cálculo en varios pasos:

1. Como hemos dicho, se presupone que hemos calculado la media  $\bar{x}$ .
2. Debemos restarle esa media  $\bar{x}$  a **cada uno** de los valores  $x_i$ .

$$(x_1 - \bar{x}), (x_2 - \bar{x}), \dots, (x_n - \bar{x}).$$

3. Después elevamos cada diferencia al cuadrado, para obtener los  $n$  valores

$$(x_1 - \bar{x})^2, (x_2 - \bar{x})^2, \dots, (x_n - \bar{x})^2.$$

4. Sumamos los cuadrados y
5. Dividimos por  $n$ .

Al utilizar la hoja de cálculo hemos ido traduciendo estos pasos de forma visual, de manera que cada uno de los pasos segundo y tercero suponían añadir una columna adicional a la hoja de cálculo. Aquí vamos a aprender cómo se hace la traducción de esos pasos al lenguaje de Python. Para empezar vamos a tratar de ilustrar la idea de una forma que recuerde a la construcción tan visual que hacíamos en la hoja de cálculo. Y al principio sólo queremos que leas: creemos que es mejor que no empieces a ejecutar código en Spyder hasta que te lo digamos.

Es bueno comenzar pensando en el objetivo que queremos alcanzar. Lo que queremos es obtener (a partir de la lista `edades` y el valor `mediaEdad`) una nueva lista, a la que vamos a llamar `diferenciasCuad` (por “*diferencias al cuadrado*”) que contenga todos los valores  $(x_i - \bar{x})^2$  cuando  $x_1, \dots, x_n$  es la lista de edades.

Inicialmente esa lista estará vacía, esperando a que la vayamos llenando de contenido. En Python se obtiene una lista vacía con dos corchetes sin nada entre ellos (o con espacios entre ellos, por razones de estilo):

```
diferenciasCuad = [ ]
```

Ahora empieza el trabajo de verdad:

1. Tomamos el primer valor de la lista `edades`. Para poder referirnos con comodidad a ese valor lo llamamos `edad`. Así que al usar el primer valor de la lista `edad` toma el valor 22. En Python es como si hicieramos:

```
edad = 22
```

2. Calculamos la diferencia:

```
diferencia = edad - mediaEdad
```

Puesto que `edad` es 22 y `mediaEdad` es 19.44 el resultado es que `diferencia` ahora vale 2.56.

3. Elevamos al cuadrado la diferencia:

```
cuadradoDif = diferencia**2
```

El resultado es que `[cuadradoDif]` vale 6.5536.

4. Ahora añadimos ese valor a la lista `diferenciasCuad`, que ahora es:

```
diferenciasCuad = [cuadradoDif]
```

¿Y ahora qué? Ahora nos toca pasar al siguiente valor de la lista `edades`, que es 21. Puedes comprobar que el siguiente fragmento de código describe los tres primeros pasos de los cuatro que hemos descrito:

```
edad = 21
diferencia = edad - mediaEdad
cuadradoDif = diferencia**2
```

Para el último paso tenemos que ser más cautos. Si hicieramos

```
diferenciasCuad = [cuadradoDif]
```

entonces estaríamos sobrescribiendo (y por tanto perdiendo) el valor anterior de la lista `diferenciasCuad`. Lo que necesitamos es concatenar el nuevo valor con la lista que ya tenemos calculada. Eso se consigue con:

```
diferenciasCuad = diferenciasCuad + [cuadradoDif]
```

De hecho, y es importante que entiendas bien este paso, puesto que inicialmente la lista `diferenciasCuad` está vacía, podríamos haber usado esta misma línea de código para el primer elemento de la lista y el resultado habría sido el mismo.

Así, el código que describe esos cuatro pasos para el segundo valor de la lista queda de este modo:

```
edad = 22
diferencia = edad - mediaEdad
cuadradoDif = diferencia**2
diferenciasCuad = diferenciasCuad + [cuadradoDif]
```

Y tras ejecutarlo la lista `diferenciasCuad` vale [6.5536, 2.4336].

Ahora pasaríamos al siguiente valor de la lista. Pero a estas alturas ya te habrás dado cuenta de que esto es muy repetitivo. Basta con cambiar el valor de `edad` en la primera línea de ese bloque de instrucciones y tras ejecutarlas habremos añadido el siguiente valor a la lista `diferenciasCuad`. Ha llegado el momento de pasar a Spyder y poner a prueba estas ideas. Para facilitar tu trabajo, hemos incluido aquí un fichero de código Python que contiene el código necesario:

### Tut02-py-BuclesFor01.py

cuyo contenido se muestra a continuación:

```
edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19,
18, 22, 20, 19, 22, 18, 20, 21, 20]

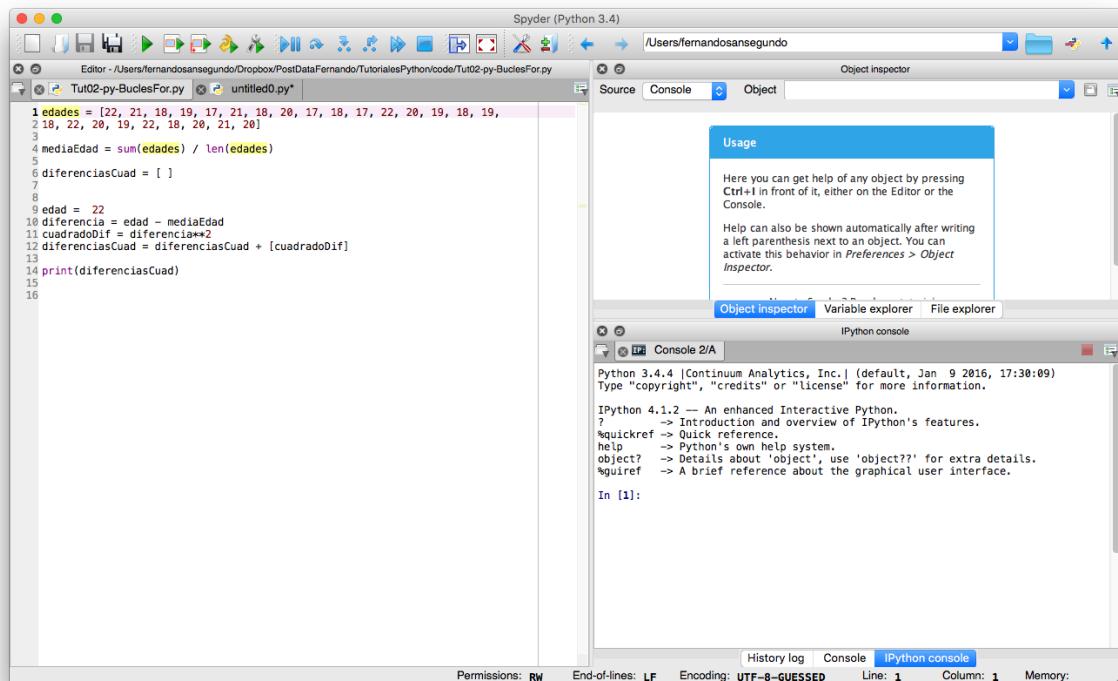
mediaEdad = sum(edades) / len(edades)

diferenciasCuad = []

edad = 22
diferencia = edad - mediaEdad
cuadradoDif = diferencia**2
diferenciasCuad = diferenciasCuad + [cuadradoDif]

print(diferenciasCuad)
```

Como verás, la lista de edades no cabía en una línea, así que la hemos repartido entre las dos primeras líneas del fichero. Pero eso no causará ningún problema. En la Sección 3 has aprendido como abrir este fichero en el *Editor de Código* de Spyder. Hazlo ahora, deberías ver algo así (se muestra la versión en Mac OS de Spyder):



Recuerda que puedes ir ejecutando las líneas de código una tras otra situando el cursor sobre ellas y pulsando F9. **¡Pero cuidado!** Como la lista no cabe en una línea, para ejecutarla tendrás que seleccionar (con el ratón o teclado) las dos líneas que ocupa antes de pulsar F9). Ve ejecutando una tras otra todas las líneas del fichero. Al terminar, tras ejecutar la línea final con `print` en la terminal debes ver este último resultado:

```
In [8]: print(diferenciasCuad)
[6.553599999999993]
```

Que es el valor inicial que esperábamos, salvo por el redondeo. Ahora cambia, en la línea 9 del código, el valor 22 por 21, que es el siguiente de la lista. A continuación selecciona las líneas de la 9 a la 12 (ambas inclusive), como muestra la figura:

```
4 mediaEdad = sum(edades) / len(edades)
5
6 diferenciasCuad = []
7
8
9 edad = 21
10 diferencia = edad - mediaEdad
11 cuadradoDif = diferencia**2
12 diferenciasCuad = diferenciasCuad + [cuadradoDif]
13
14 print(diferenciasCuad)
15
```

y pulsa otra vez F9. En la *Terminal* de Spyder verás que se ha ejecutado ese bloque de código, pero no aparece ninguna salida. Para ver el resultado sitúate en la línea 14 del *Editor*, la que contiene la función `print`, y pulsa F9. El resultado que verás en la *Terminal* es:

```
In [10]: print(diferenciasCuad)
[6.553599999999993, 2.433599999999996]
```

Una última vez: cambia el valor 21 por 18 en la línea 9 y repite los pasos anteriores hasta llegar a

```
In [12]: print(diferenciasCuad)
[6.553599999999993, 2.433599999999996, 2.073600000000004]
```

¿Va quedando claro? Naturalmente, podrías seguir así cambiando cada vez la línea 9 por el siguiente elemento de la lista `edades`. Pero hacer eso a mano es una tarea muy repetitiva y aburrida. Que, además, se convertiría en inviable si la lista tuviera cientos de valores. Sería mucho mejor que hubiera una forma de pedirle a Python que se encargara de esta tarea: “*Oye, Python: ve asignando a la variable edad cada uno de los valores de edades y para cada una de esas asignaciones, ejecuta las líneas de la 10 a la 12.*” Pues estamos de enhorabuena, porque eso es exactamente lo que hace el bucle `for`. Y el código es muy parecido a lo que ya tenemos. Concretamente, el código está en el fichero adjunto:

[Tut02-py-BucleFor02.py](#)

cuyo contenido se muestra a continuación:

```
edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19,
18, 22, 20, 19, 22, 18, 20, 21, 20]

mediaEdad = sum(edades) / len(edades)

diferenciasCuad = []

for edad in edades:
    diferencia = edad - mediaEdad
    cuadradoDif = diferencia**2
    diferenciasCuad = diferenciasCuad + [cuadradoDif]

print(diferenciasCuad)
```

Abre este fichero en el *Editor de Código* de Spyder (puedes cerrar el anterior si quieras y no hace falta que guardes los cambios que hemos hecho). Asegúrate además de hacer limpieza en la *Terminal* ejecutando `%reset` y después `%clear`. Una vez hecho esto, sitúate en el *Editor de Código* (en cualquier punto) y pulsa F5 para ejecutar todo el programa a la vez. Al hacerlo verá aparecer la salida en la *Terminal*. Como ya sabes, Python ha usado la función `runfile` para ejecutar nuestro código y el resultado es esta lista de números:

```
[6.553599999999993, 2.433599999999996, 2.073600000000004, 0.1936000000000113, 5.953600000000006, 2.433599999999996, 2.073600000000004, 0.3135999999999855, 5.953600000000006, 2.073600000000004, 5.953600000000006, 6.553599999999993, 0.3135999999999855, 0.1936000000000113, 2.073600000000004, 6.553599999999993, 0.3135999999999855, 0.1936000000000113, 6.553599999999993, 2.073600000000004, 0.3135999999999855, 2.433599999999996, 0.3135999999999855]
```

que son los 25 valores que contiene la lista `diferenciasCuad` al terminar de ejecutarse el código. Reconocerás algunos de los primeros valores porque son los que obtuvimos con el anterior programa. Como ves, este código hace lo que queríamos y construye la lista de diferencias cuadráticas con respecto a la media para la lista `edades`.

Vamos con el análisis del código propiamente dicho. La diferencia con el anterior fichero está en las líneas de la 9 a la 12, que ahora forman el bucle `for`. Fíjate en estos detalles sobre la estructura de ese bloque:

1. La primera línea del bucle, que es

```
for edad in edades:
```

es la línea de cabecera del bucle `for`. La variable `edad` que aparece aquí es la variable índice del bucle, cuyo papel consiste en ir recorriendo uno a uno los valores de la lista `edades`. Podríamos haber usado cualquier otro nombre para esa variable índice (volveremos sobre esto enseguida). Un **detalle muy importante** es que esa línea termina con dos puntos. Si los olvidas, Python señalará un error en esa línea.

2. Las tres siguientes líneas de código

```
diferencia = edad - mediaEdad  
cuadradoDif = diferencia**2  
diferenciasCuad = diferenciasCuad + [cuadradoDif]
```

forman el cuerpo del bucle `for`. Estas líneas se ejecutan una vez para cada valor que toma la variable índice del bucle. Es decir, una vez para cada uno de los elementos de la lista `edades`. Cada una de esas ejecuciones se denomina una iteración del bucle. En la primera iteración del bucle, la variable auxiliar `edad` toma el primer valor de la lista `edades` (es decir, 22). En la segunda iteración, `edad` toma el segundo valor de la lista `edades` (que es 21), etc.

Fíjate en **otro detalle muy importante**. Esas líneas de código están indentadas; es decir, que hemos usado espacios en blanco al principio de las líneas para desplazarlas a la derecha con respecto a la línea de cabecera del bucle. En Python todas las líneas que forman el cuerpo de un bucle `for` deben estar indentadas con respecto a la línea de cabecera. De hecho esa indentación es la forma (la *única* forma) que usa Python para saber donde empieza y donde termina el cuerpo del código (en otros lenguajes el cuerpo del bucle se encierra, por ejemplo, entre llaves).

3. La última línea del programa:

```
print(diferenciasCuad)
```

no está indentada, de manera que ya no forma parte del bucle `for`. Eso significa que esa línea de código se ejecuta sólo una vez, cuando han terminado todas las iteraciones del bucle `for`. Por esa razón la salida del programa sólo muestra la lista `diferenciasCuad` completa.

Para tratar de entender un poco mejor algunos aspectos del funcionamiento del bucle `for` vamos a hacer un ejercicio.

### Ejercicio 15.

1. Prueba a eliminar los dos puntos del final de la línea de cabecera del bucle y ejecuta otra vez todo el bloque de código. ¿Cuál es el mensaje de error?
2. Vamos a ejecutar otra vez el bucle `for`, pero cambiando el nombre de la variable auxiliar `edad` por `x`. Las líneas de la 9 a la 12 (las del bucle) deben quedar así:

```

for x in edades:
    diferencia = x - mediaEdad
    cuadradoDif = diferencia**2
    diferenciasCuad = diferenciasCuad + [cuadradoDif]

```

*Hemos cambiado por **x** las dos apariciones de **edad**. ¿Hay algún cambio en el resultado final? Prueba a usar otro nombre cualquiera y repite este apartado con ese nombre. En cualquier caso, a pesar de la tentación de usar nombres cortos como **x** para escribir menos, recuerda que es conveniente que el nombre sea descriptivo del papel que juega la variable.*

3. Vamos a modificar otra vez el programa. Volvemos a usar **edad** para la variable índice del bucle, pero ahora además cortamos y pegamos la línea de la función **print** para que la parte final del programa quede así (se muestran las líneas de la 9 en adelante; no hay más código por debajo):

*¿Qué sucede ahora al ejecutar el código?*



### Otra forma de expresar un bucle: comprensión de listas.

Los bucles **for** como los que hemos visto en el apartado anterior existen en muchos otros lenguajes de programación, con prácticamente la misma estructura. Sólo varían los detalles de formato propios de cada lenguaje. Pero Python dispone además de una forma alternativa de expresar bucles; es decir, de ejecutar la misma colección de operaciones sobre los elementos de una lista. Esta segunda manera de construir los bucles propia de Python puede resultarte un poco más difícil de entender al principio, porque toda la estructura del bucle se condensa en una única línea de código. Por ejemplo, el cálculo de las desviaciones al cuadrado de cada elemento de **edades** se puede obtener de esta forma alternativa así:

```
diferenciasCuad = [(edad - mediaEdad)**2 for edad in edades]
```

Enseguida vamos a entrar en detalles sobre esta nueva forma de expresar el bucle. Pero antes vamos a comprobar que esta línea tiene el mismo efecto que el bucle **for** que vimos en el programa **Tut02-py-BuclesFor02.py** de la anterior sección. Para ello en el siguiente programa adjunto hemos sustituido todo el bucle **for** (incluida la línea en la que creábamos una lista vacía) con esa única línea.

[Tut02-py-ComprensionLista01.py](#)

El contenido del fichero se muestra a continuación:

```

edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19,
18, 22, 20, 19, 22, 18, 20, 21, 20]

mediaEdad = sum(edades) / len(edades)

diferenciasCuad = [(edad - mediaEdad)**2 for edad in edades]

print(diferenciasCuad)

```

Como hemos hecho otras veces, abre este fichero en el *Editor de Código* de Spyder, asegurándote de que estás en una sesión limpia (recuerda **%reset** y **%clear**). Ejecútalo (recuerda que F5 ejecuta todo el programa) y podrás comprobar que el resultado es el mismo que al usar el bucle **for**. Esta segunda forma de trabajar se denomina **comprensión de listas**, una traducción decepcionantemente literal del nombre en inglés *list comprehension* (que ya es, en sí mismo, desafortunado).

La comprensión de lista en nuestro ejemplo se refiere concretamente a la expresión entre corchetes:

```
[(edad - mediaEdad)**2 for edad in edades]
```

Esta expresión se puede ver como una receta para fabricar una lista: es como si le dijéramos a Python:

```
[haz esta operación para cada elemento de la lista]
```

Y de nuevo, como sucedía en el bucle `for`, la variable auxiliar puede recibir cualquier nombre y el resultado es el mismo. Pruebas a reemplazar la comprensión de listas del programa con esta otra:

```
[(item - mediaEdad)**2 for item in edades]
```

Puesto que en inglés *un elemento de la lista* se suele escribir *an item in the list*, es frecuente que los programadores de Python usen `item` como nombre para la variable auxiliar cuando no hay un nombre preferible. Recuerda en cualquier caso lo que hemos dicho sobre la conveniencia de usar nombres de variable esclarecedores.

### Ejercicio 16.

1. *Si no lo has hecho ya, ejecuta este código con `item` para comprobar que en efecto produce el mismo resultado.*
2. *Ya tenemos dos versiones del bucle (la versión `for` y la comprensión de listas) pero en realidad no hemos terminado el cálculo de la varianza poblacional de `edades`. Completa el cálculo usando la versión del bucle que prefieras (da igual, claro).*

□

La principal ventaja de la comprensión de listas frente al bucle `for` es su concisión. Su principal inconveniente es la pérdida de claridad del código que puede llevar aparejada esa misma concisión. En este y en los próximos Tutoriales tendrás ocasión de ver muchos ejemplos de comprensión de listas. Así que si ahora no acabas de entender su funcionamiento, ten un poco de paciencia. Irá quedando más y más claro con la práctica.

## 5. Recursos para facilitar el trabajo: ordenación y `print` con formato.

Los temas que hemos agrupado en esta sección del tutorial no tienen un hilo temático claro, como sucede en otras secciones. Pero son recursos que van a hacer mucho más sencillo parte del trabajo que tenemos por delante y los vamos a necesitar pronto. Sirven, en cualquier caso, para incluir algunos ejemplos que aumentan nuestra experiencia con el código en Python.

### 5.1. Ordenación de una lista: `in situ` vs externa.

**Advertencia previa:** En las secciones previas hemos ido mostrando como trabajar en el *Editor de Código* y la *Terminal* de Spyder. Pero a partir de este apartado, vamos a ser menos específicos en las instrucciones. A menudo diremos cosas como “*Ejecuta este código*” y mencionaremos (o no) la *Terminal* o el *Editor*. En muchos casos, eso significa que estamos haciendo experimentos con alguna idea nueva. Y la decisión sobre la forma que te resulta más cómoda de trabajar es tuya. A veces preferirás copiar el código en el *Editor* para que te sea más cómodo hacer cambios y ejecutarlo con `F5` o `F9`. Otras veces irás directamente a la *Terminal*. Y, eso sí, cuando sea necesario te daremos instrucciones explícitas sobre la forma en la que debes ejecutar el código. En cualquier caso, el mejor consejo que podemos darte es que hagas pruebas y pruebas (nosotros también seguimos buscando cada día formas mejores de trabajar). Recuerda, eso sí, que si quieres evitar interferencias del trabajo que has hecho en sesiones previas, debes asegurarte de trabajar en una *sesión limpia*.

Vamos a empezar con el trabajo de esta sección. El primer problema que queremos abordar es el de la ordenación de una lista. Y el caso más habitual es el de una lista de números. Por ejemplo, dada esta lista no ordenada:

```
numeros = [10, 8, 43, 7, 24, 7, 31, 45, 1, 3, 20, 14, 12, 44, 13, 20, 33, 7, 29, 5]
```

podemos ordenarla fácilmente con la función `sorted`. Esto es lo que obtenemos en la *Terminal* de Spyder:

```
In [1]: numeros = [10, 8, 43, 7, 24, 7, 31, 45, 1, 3, 20, 14, 12, 44, 13, 20, 33, 7, 29, 5]
In [2]: print(sorted(numeros))
[1, 3, 5, 7, 7, 7, 8, 10, 12, 13, 14, 20, 20, 24, 29, 31, 33, 43, 44, 45]
```

Si lo que queremos es ordenarlos de mayor a menor basta con añadir un argumento a la función:

```
In [3]: print(sorted(numeros, reverse=True))
[45, 44, 43, 33, 31, 29, 24, 20, 20, 14, 13, 12, 10, 8, 7, 7, 5, 3, 1]
```

El valor `True` es uno de los dos valores booleanos de Python, `True/False` (cierto/falso) sobre los que volveremos más adelante. De momento puedes pensar en el argumento `reverse=True` como un interruptor que permite activar o desactivar el orden decreciente en la función `sorted`. Iremos viendo que muchas otras funciones de Python tienen argumentos booleanos como este que sirven precisamente para comutar entre dos posibles comportamientos de la función.

Fíjate en que el proceso de ordenación no ha afectado a la lista original, que sigue desordenada:

```
In [4]: print(numeros)
[10, 8, 43, 7, 24, 7, 31, 45, 1, 3, 20, 14, 12, 44, 13, 20, 33, 7, 29, 5]
```

Por eso decimos que la función `sorted` hace una ordenación externa. En otras ocasiones preferiremos que la lista ordenada reemplace a la lista original. Hay dos formas de hacer esto y es probable que ya hayas adivinado cuál es la primera. Bastaría con hacer:

```
numeros = sorted(numeros)
```

Pero no vamos a hacer esto, porque queremos aprovechar para mostrarte el segundo procedimiento y de paso aprender un poco más de Python. El segundo método consiste en ejecutar el siguiente comando (el paréntesis vacío es imprescindible):

```
numeros.sort()
```

Vamos a hacer esto en la *Terminal*, mostrando la lista `numeros` antes y después de ejecutar ese comando:

```
In [5]: print(numeros)
[10, 8, 43, 7, 24, 7, 31, 45, 1, 3, 20, 14, 12, 44, 13, 20, 33, 7, 29, 5]

In [6]: numeros.sort()

In [7]: print(numeros)
[1, 3, 5, 7, 7, 7, 8, 10, 12, 13, 14, 20, 20, 24, 29, 31, 33, 43, 44, 45]
```

El resultado es el que queríamos: la lista ordenada remplaza a la original. Esto es lo que se conoce como ordenación *in situ*. En general usamos ese término cuando una modificación de un objeto ocupa el lugar del objeto original.

Pero el otro aspecto interesante de este segunda manera de ordenar la lista es el propio formato del comando que hemos usado. Python es un lenguaje orientado a objetos. Aunque no vamos a entrar en la discusión técnica de lo que eso significa en Computación, sí queremos que te familiarices con algunos consecuencias prácticas de esa característica de Python. Todas las construcciones que vamos viendo: variables, listas, funciones y muchas otras que veremos son objetos. Y cada objeto de Python tiene un serie de métodos asociados. Los métodos representan acciones que podemos llevar a cabo usando ese objeto. Por ejemplo, cualquier objeto de clase *lista* (como `numeros`) tiene asociado el método `sort` que permite ordenar *in situ* ese objeto. La forma general de invocar un método en Python es un comando de la forma:

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

El comando `numeros.sort()` que hemos visto es un ejemplo de esta construcción, aunque en ese caso el método `sort` se invoca sin argumentos (de ahí el paréntesis vacío; prueba a eliminarlo y mira lo que sucede). También podríamos haber hecho ordenación *in situ* descendente:

```
In [9]: numeros.sort(reverse=True)

In [10]: print(numeros)
[45, 44, 43, 33, 31, 29, 24, 20, 20, 14, 13, 12, 10, 8, 7, 7, 7, 5, 3, 1]
```

y en este caso el método `sort` si tiene un argumento que es: `reverse=True`. A lo largo del curso nos vamos a encontrar muchas veces con dos formas de ejecutar acciones en Python. La primera que vimos es de la forma:

```
funcion(argumentos)
```

y la que estamos presentando ahora, que es:

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

Ambas son comunes a casi todos los lenguajes de programación modernos.

Antes de cerrar este apartado queremos señalar que también podemos usar estos métodos para la ordenación alfabética de una lista de cadenas de caracteres. Por ejemplo partiendo de esta lista:

```
meses = ["enero", "febrero", "marzo", "abril", "mayo", "junio", "julio", "agosto",
         "septiembre", "octubre", "noviembre", "diciembre"]
```

¿qué sucede si ejecutas el siguiente código?

```
mesesOrd = sorted(meses)
print(mesesOrd)
```

## 5.2. De nuevo la función `print`. Textos con formato.

Ya dijimos al presentarla que la función `print` nos proporciona un mecanismo de control mucho más fino sobre la forma de mostrar los resultados de nuestro código. En este apartado vamos a ver como combinar la función `print` con las variables de tipo cadena y los bucles y rangos para dar un salto cualitativo en nuestra capacidad de expresarnos mediante Python.

Empezamos con un ejemplo sencillo. Fíjate en lo que sucede al ejecutar este código en la *Terminal*:

```
In [1]: a = 7

In [2]: print("El valor de la variable a es {}" .format(a))
El valor de la variable a es 7
```

El resultado es que la función `print` produce como salida la cadena de caracteres, pero al hacerlo sustituye la parte `{0}` de esa cadena con el valor de la variable `a`. ¿Y cómo sabe Python cuál es la variable que debe usar como sustituto de `{0}`? Se lo hemos indicado mediante el método `format` aplicado a esa cadena de caracteres. El mecanismo puede resultar un poco lioso al principio, pero con la práctica resulta más natural y es en cualquier caso una herramienta de presentación muy potente, como tendremos ocasión de comprobar en estos tutoriales. Veamos otros dos ejemplos que introducen novedades interesantes:

```
In [1]: tiempo = 7

In [2]: espacio = 123

In [3]: velocidad = espacio / tiempo

In [4]: print("Hemos recorrido {} metros en {} segundos." .format(espacio, tiempo))
Hemos recorrido 123 metros en 7 segundos.

In [5]: print("Por lo tanto la velocidad ha sido igual a {:.2f} m/s" .format(velocidad))
Por lo tanto la velocidad ha sido igual a 17.57 m/s
```

El primer uso de la función `print` es muy parecido al ejemplo anterior. La novedad es que aparecen dos variables en vez de una, y por tanto hemos usado `{0}` y `{1}` para indicarle a `print` cuál es la variable que debe sustituir en cada posición (recuerda siempre que Python cuenta desde 0). Despu s el m todo `format` le proporciona a `print` esas variables, que se usan en el orden en el que aparecen como argumentos de `format`.

Todas las variables que hemos usado en los ejemplos previos eran de tipo `int`. En cambio, la segunda llamada a `print` de este ejemplo utiliza s lo una variable, pero esa variable es de tipo `float`. Adem s, las instrucciones que usamos para pedirle a Python que sustituya el valor de esa variable son m s complicadas: hemos usado `{0:5.2f}`. ¿Qu  significa esto? La expresi n entre llaves tiene dos partes, separadas por los dos puntos. La primera parte es simplemente el n mero de orden de la variable para el caso en que haya m s de una y se corresponde, como hemos visto, con el orden en el que aparecer n enumeradas las variables en la llamada a `format`. En este caso aparece un 0, porque s lo hay que sustituir la variable `velocidad`. La parte que sigue a los dos puntos `5.2f` es nueva. Empecemos por lo m s f cil: la letra `f` le indica a Python que se trata de sustituir una variable de tipo `float`. Una vez aclarado eso, el s mbolo `5.2` significa: “usa 5 espacios, y muestra dos decimales despu s de la coma”. Vamos a hacer una modificaci n en esos valores para ver el efecto:

```
In [6]: print("Por lo tanto la velocidad ha sido igual a :{0:15.4f} m/s".format(velocidad))
Por lo tanto la velocidad ha sido igual a           17.5714 m/s
```

F jate en que al usar `15.4f` ahora aparecen cuatro cifras decimales despu s de la coma. Adem s ese espacio en blanco que ha aparecido antes del valor de la variable se debe a que le hemos pedido a Python que use 15 espacios para mostrar el valor de la variable. Y puesto que no necesitaba tantos, una parte de ellos est n en blanco. M s adelante veremos que esto puede ser muy \'util para dar un formato visual conveniente a nuestros resultados; por ejemplo al imprimir una tabla de valores, en la que queremos controlar la anchura de cada columna. Lo veremos a continuaci n.

### C mo imprimir una tabla de valores.

La comprensi n de listas, como hemos visto, es un proceso iterativo que sirve para fabricar una lista elemento a elemento. Por ejemplo, podemos usarla para fabricar una lista que represente una tabla de valores. Imag ntate que vas a viajar en breve al Reino Unido y que tu moneda local es el euro. Puesto que all  usan la libra esterlina, puede resultar conveniente fabricar una tabla de conversi n de precios en libras a precios en euros, que te permita por ejemplo saber si te est n cobrando un precio exorbitante por esa pinta de cerveza. En el momento de escribir este tutorial, a comienzos del a o 2016, el tipo de cambio es:

*Una libra equivale a 1.3164 euros.*

Vamos a fabricar una tabla que convierta los precios en libras, de media libra en media libra, desde 0.5 hasta 10 libras. Enseguida ver s que son 20 valores. As  que podemos fabricar los valores en libras usando `range(1:21)` y la comprensi n de listas as :

```
In [21]: libras = [valor * 0.5 for valor in range(1, 21)]
In [22]: print(libras)
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5,
8.0, 8.5, 9.0, 9.5, 10.0]
```

Para convertir estas cantidades en libras a euros podemos usar otra vez una comprensi n de lista. Vamos a introducir el tipo de cambio en una variable para que el c digo sea m s f cil de entender y de modificar si el tipo de cambio sufre alguna alteraci n. :

```
In [23]: tipoCambio = 1.3164
In [24]: euros = [valor * tipoCambio for valor in libras]
In [25]: print(euros)
[0.6582, 1.3164, 1.9746000000000001, 2.6328, 3.291, 3.949200000000003, 4.6074,
5.2656, 5.9238, 6.582, 7.2402, 7.898400000000005, 8.5566, 9.2148, 9.873, 10.5312,
11.18940000000001, 11.8476, 12.5058, 13.164]
```

Podríamos conformarnos con este resultado. Pero el resultado no es muy cómodo, ni fácil de emplear en la práctica. Sería mucho mejor presentar nuestros resultados en una tabla. Y aquí tenemos una ocasión para comparar la comprensión de listas con el bucle `for`. Como hemos visto, la comprensión de listas sirve para fabricar elementos iterativamente. Pero para fabricar la tabla queremos usar la función `print` varias veces, una por cada línea. Y cuando se trata de *repetir acciones* a menudo es más natural expresar esa repetición mediante un bucle `for`. El código para fabricar la tabla podría ser este, que usa los formatos de `print` que hemos visto antes:

```
print("Libras | Euros")
print("-----|-----")
for i in range(0, 20):
    print(" {0:4.1f} | {1:5.2f}".format(libras[i], euros[i]))
    print("-----|-----")
```

El resultado de ejecutar este código en la *Terminal* aparece en la Tabla 1 (pág. 45). El resultado es, desde luego, una tabla mucho más fácil de usar. Para conseguir ajustar el formato de esa tabla hemos tenido que hacer algo de ensayo y error con los espacios y el número de cifras decimales, pero son manipulaciones sencillas que tú mismo podrás experimentar en futuros ejemplos. Es cierto que ese formato no es impresionante, pero a partir de aquí, cualquier programador con unos conocimientos básicos de lenguajes para la Web (basta con los rudimentos de HTML y CSS) podría fácilmente escribir un programa que fabricaría una página web con esta tabla y el estilo que se desee (tipografías, colores, fondos, etc.) Es más, con apenas un poco más de aprendizaje de Python sería fácil diseñar un programa que cada cierto tiempo obtuviera la tasa de cambio libras/euros desde un servidor de Internet y la usara para actualizar una página web con una tabla de cambios como esta. Tú mismo puedes imaginarte muchas otras aplicaciones similares. La presentación automatizada de resultados de análisis estadísticos mediante tablas o gráficos (a menudo el resultado se diseña en formato web) es una parte fundamental de la visualización que sirve de base a la comunicación científico-técnica actual.

**Ejercicio 17.** ¿Por qué al fabricar la lista `libras` hemos usado `range(1:21)`? Y teniendo esto en cuenta, ¿por qué en el bucle `for` de la tabla hemos usado `range(0:20)`?

□

Este ejercicio apunta a una situación frecuente en programación, cuando queremos iterar *tomando como referencia una lista*. Algo así como: “*repita esto tantas veces como elementos tiene una lista dada*.” La forma en la que hemos resuelto esto aquí es un poco artificiosa, veremos más adelante en el curso maneras mejores (más “pythónicas”, como suele decirse) de abordar este problema.

### Más formatos para `print` y cifras significativas.

Para cerrar nuestro primer encuentro con las posibilidades que ofrece `print` combinada el método `format`, queremos añadir algunos comentarios sobre las opciones disponibles para formatear valores numéricos. Hemos visto que podemos usar una construcción como `{0:5.2f}` para indicarle a `print` que queremos mostrar un número usando cinco espacios y dos cifras tras la coma. Existen otras construcciones similares, sustituyendo `f` por otros códigos. Por ejemplo, si usamos `{0:5.2e}` mira lo que se obtiene al pedirle a `print` que nos muestre el valor de `pi`:

```
In [1]: import math as m
In [2]: print("{0:5.2e}".format(m.pi))
3.14e+00
```

El resultado es que el número se muestra en notación científica, y que 5.2 se utiliza para controlar tanto el número de posiciones que ocupa el número como el número de cifras tras la coma decimal. Estos otros ejemplos pueden aclarar cómo funciona esto. En cada versión hemos modificado cada componente del formato unidad a unidad para que puedas ver el efecto:

```
In [3]: a = 163.5735
In [4]: print("{0:9.3e}".format(a))
1.636e+02
```

Libras	Euros
0.5	0.66
1.0	1.32
1.5	1.97
2.0	2.63
2.5	3.29
3.0	3.95
3.5	4.61
4.0	5.27
4.5	5.92
5.0	6.58
5.5	7.24
6.0	7.90
6.5	8.56
7.0	9.21
7.5	9.87
8.0	10.53
8.5	11.19
9.0	11.85
9.5	12.51
10.0	13.16

Tabla 1: Un ejemplo de tabla (cambio de libras esterlinas a euros).

```
In [5]: print("{0:10.3e}".format(a))
1.636e+02

In [6]: print("{0:10.4e}".format(a))
1.6357e+02

In [7]: print("{0:11.4e}".format(a))
1.6357e+02
```

Prueba con otros valores hasta convencerte de que entiendes lo que sucede. Aparte de `f` y `e`, existen muchos otros códigos de formato. La documentación oficial aparece en este enlace:

<https://docs.python.org/2/library/string.html#format-specification-mini-language>

y si lo visitas podrás comprobar que apenas nos hemos asomado al tema de los formatos disponibles. Antes de seguir adelante sólo queremos añadir que el código de formato `g` permite seleccionar el número de cifras significativas con las que se muestra un número. Por ejemplo supongamos que, como en la Sección 1.3 del libro (pág. 15) queremos mostrar el número

```
In [1]: a = 1.623698
```

con cuatro cifras significativas. Para ello basta con hacer:

```
In [2]: print("{0:.4g}".format(a))
1.624
```

y se obtiene el resultado. Fíjate en que no es necesario indicar el número de espacios, Python lo asignan automáticamente si no lo incluimos. Siguiendo con el ejemplo más complicado de esa sección, para mostrar el número

```
In [3]: b = 0.00337995246
```

con cinco cifras significativas hacemos:

```
In [121]: print("{0:.5g}".format(b))
0.00338
```

En este caso conviene observar que Python, al igual que otros lenguajes, desgraciadamente no incluye ceros a la izquierda en este tipo de redondeos.

## 6. Comentarios.

Hemos insistido ya varias veces en que la legibilidad del código es crucial y debe ser una preocupación constante de cualquier programador. Es algo que se debe aprender desde el principio, para incorporarlo a tu conjunto de buenas prácticas científicas. En el trabajo científico la colaboración entre investigadores y la difusión del conocimiento juegan un papel crucial. Y puesto que la computación es en la actualidad una parte insoslayable de ese trabajo, es cada vez más común y necesario compartir con otras personas el código que escribimos. A la vez que naturalmente nos convertimos en receptores y usuarios de código escrito por otros. Con esa idea en mente, la legibilidad del código resulta ser una necesidad imperiosa. Y la elección de nombres adecuados para las variables es sólo el primer paso. Lo que realmente se necesita es una buena documentación del código.

Documentar significa, en el contexto de los ficheros de código, añadir a esos ficheros información que no está pensada para dar instrucciones al ordenador, sino que ha sido pensada para ayudarnos a entender lo que se está haciendo en ese programa. Es decir, esa información no es para la máquina. Es para nosotros mismos, o para otros usuarios (humanos) de ese fichero. A lo largo del curso, en los tutoriales, nosotros te vamos a facilitar una serie de ficheros (los llamaremos “*plantillas*”), que contienen código preparado para llevar a cabo algunos de los métodos que vamos a aprender en cada capítulo del curso. Cuando abras por primera vez uno de esos ficheros, y especialmente al tratarse de métodos con los que, aún, no estás familiarizado, necesitarás sin duda unas “instrucciones de manejo”, para saber cómo utilizar el fichero. Esas instrucciones podrían ir en un fichero aparte, claro. Pero la experiencia ha demostrado que esa no es una buena manera de

organizar el trabajo. Si el código y la documentación van por separado, es casi inevitable que, al final, tras algunos cambios, ya no se correspondan, y la documentación pase a ser contraproducente. Afortunadamente, a lo largo del tiempo se han desarrollado una serie de métodos para garantizar una correcta documentación del código, que van desde los más sencillos hasta ideas sofisticadas como el control de versiones y la programación literaria. Aquí vamos a empezar por la manera más sencilla de combinar código y documentación. Más adelante tal vez necesites métodos más sofisticados, pero esos métodos se añadirán a lo que vamos a aprender aquí, sin remplazarlo.

La idea básica es que cuando usamos el símbolo `#` en una línea de código, Python ignora todo el código que aparezca en esa línea a la derecha del símbolo `#`. Vamos a comprobar esto. Hemos ejecutado en la terminal de IPython los siguientes comandos:

```
In [1]: a = 1  
In [2]: a = a + 2  
In [3]: print(a)  
3
```

¿Todo normal, verdad? Ahora, para empezar desde cero, he reiniciado la terminal de IPython. Puedes cerrar Spyder y abrirlo de nuevo, o puedes usar la función mágica `%reset`. En cualquier caso, ahora ejecutamos el mismo código pero introduciendo el símbolo `#` al principio de la segunda fila:

```
In [4]: %reset  
Once deleted, variables cannot be recovered. Proceed (y/[n])? y  
In [5]: a = 1  
In [6]: # a = a + 2  
In [7]: print(a)  
1
```

Si haces esto verás que desde el mismo momento en que escribes `#` la línea cambia de aspecto. IPython te está indicando visualmente que ese código no se va a ejecutar. Y el resultado confirma que Python ha ignorado esa línea. Vamos a practicar esto en un ejercicio:

**Ejercicio 18.** ¿Qué va a ocurrir al ejecutar estas tres versiones del código? Trata de adivinarlo antes de hacerlo.

```
1. %reset  
a = 1  
a = a # + 2  
print(a)
```

```
2. %reset  
a = 1  
a = a + # 2  
print(a)
```

```
3. %reset  
a = 1  
a = a + 2 # Este caso es interesante.  
print(a)
```



El último caso de este ejercicio es, en efecto, interesante. Ese ejemplo muestra como podemos utilizar el símbolo `#` para introducir comentarios en medio del código Python. Esas líneas de comentario son extremadamente útiles para explicar lo que está sucediendo en el código, el papel que juegan las variables o para dar al usuario instrucciones precisas sobre el funcionamiento del programa. Para insistir en esta idea vamos a mostrarte dos versiones de un programa, que simplemente calcula la varianza y desviación típica poblaciones de la lista de edades que hemos usado de ejemplo en secciones previas. La primera versión no sigue ninguna de nuestras recomendaciones en cuanto a estilo y documentación. Puedes encontrarla en el fichero adjunto:

[Tut02-mediaVarianza-01.py](#)

cuyo contenido se muestra a continuación:

```
edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22,
20, 19, 22, 18, 20, 21, 20]
m = sum(edades) / len(edades)
print(m)
nV = []
for edad in edades:
    nV = nV + [(edad - m)**2]
v = sum(nV) / len(edades)
print(v)
import math as m
print(m.sqrt(v))
```

Abre este fichero en el *Editor de Código* de Spyder y ejecútalo. Como podrás comprobar, el código funciona correctamente y muestra como resultados los valores que queríamos obtener. Por orden, la media, varianza y desviación típica poblacionales. Ahora veamos la segunda versión, que está en el fichero adjunto:

[Tut02-mediaVarianza-02.py](#)

y que se muestra a continuación:

```
"""
www.postdata -statistics.com
POSTDATA. Introducción a la Estadística.
Tutorial 02 (versión Python).
Ejemplo de cálculo de media, varianza y desv. típica
para una variable cuantitativa, datos no agrupados.
"""

import math as m

# Esta es la lista de datos sobre la que vamos a trabajar:
edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22,
20, 19, 22, 18, 20, 21, 20]

# La lista se muestra en pantalla:
print("La lista de edades es:")
print(edades)

# Media aritmética de la lista:
mediaEdad = sum(edades) / len(edades)
print("La media aritmética es:")
print(mediaEdad)

# Varianza poblacional de la lista
diferenciasCuad = []
# diferenciasCuad acumula resultados parciales del numerador de la
# varianza en cada iteración del siguiente bucle for.
for edad in edades:
```

```

diferencia = edad - mediaEdad
cuadradoDif = diferencia**2
diferenciasCuad = diferenciasCuad + [cuadradoDif]
varianzaPob = sum(diferenciasCuad) / len(edades)
print("La varianza poblacional es:")
print(varianzaPob)

# Desviación típica poblacional.
print("La desviación típica poblacional es:")
print(m.sqrt(varianzaPob))

```

Es un fichero más largo. Como antes, abre el fichero en Spyder y ejecútalo. Los resultados numéricos son, desde luego, los mismos. Pero ahora además hemos usado varias sentencias `print` para indicar cuál es la relación de cada uno de esos valores con los datos. Desde el punto de vista de un *usuario* del programa, la diferencia es enorme. Ahora vuelve a examinar comparándolos el código de ambos programas. Para motivarte en esa comparación, imagínate que te han enviado el programa por correo electrónico, sin más explicaciones, y quieres entender lo que hace el programa. En la primera versión no hay más ayuda que tus conocimientos de Python y Estadística. Ni siquiera los nombres de las variables ayudan demasiado a entender que está ocurriendo; hemos usado `v` por varianza y `nV` por *numerador de la varianza*, pero el destinatario del código tendría que adivinar eso. En la segunda versión nos han puesto las cosas mucho más fáciles mediante una serie de recursos de documentación:

- Las primeras líneas del programa son un bloque de comentarios, delimitado por dos líneas que contienen tres comillas dobles """ . Este tipo de bloques de comentario, que no habíamos visto hasta ahora, se utilizan en Python para introducir comentarios que ocupan más de una línea y son típicos de estas labores de documentación del código. En este bloque proporcionamos información sobre el programa, su procedencia, autoría (con algún tipo de información de contacto, para poder resolver dudas por ejemplo) y el objetivo del código. Es bueno acostumbrarse a incluir ese tipo de información en nuestros programas.
- A continuación hemos importado el módulo math con el alias `m`. Una de las recomendaciones de estilo que forman parte de las buenas prácticas recomendadas al escribir código Python consiste en colocar todas importaciones de módulos al principio del código, para que sean fáciles de localizar.
- Además, como ves, hemos usado líneas en blanco para dividir el programa en bloques lógicos, donde cada bloque de código persigue una finalidad concreta y diferenciada del resto del programa. Cada bloque comienza con una línea o más de comentarios `#` que describe lo que sucede en ese bloque. Esas líneas en blanco y comentarios no tienen ningún efecto a la hora de ejecutar el programa, pero ayudan a la legibilidad y a hacer explícita la estructura lógica del programa.
- En algunos casos se añaden líneas de comentario adicionales dentro del bloque de código, como hemos hecho en el caso del bucle `for` de cálculo de la varianza poblacional, si creemos que eso es necesario para ayudar al lector del programa a entender un punto concreto.
- Y desde luego, los nombres de las variables se han elegido de manera que puedan resultar informativos del papel que juegan en el programa.

El resultado, si además contamos con la ayuda de reconocimiento de sintaxis que nos presta un programa como Spyder, es un programa del que resulta mucho más fácil entender la estructura y finalidad, un programa más legible. Es importante incorporar esa disciplina a nuestro método de trabajo y pensar *siempre* que escribimos nuestros programas para que los pueda leer un lector humano. Que a menudo seremos nosotros mismos, al cabo de un cierto tiempo, esforzándonos para entender cómo funcionaba ese programa del que somos autores y que ahora nos parece ajeno. Al principio, especialmente al escribir programas cortos, es posible que pienses que es una pérdida de tiempo introducir tantos comentarios y prestar tanta atención a la documentación del código. Créenos: la forma más segura de perder el tiempo es no hacerlo.

Hemos hablado en el párrafo anterior sobre las recomendaciones de estilo en programas Python. Es un poco prematuro profundizar en esas normas cuando hemos avanzado tan poco todavía, pero para que nos sirva de referencia aquí tienes un enlace a un documento en el que se explicitan algunas de esas recomendaciones de estilo:

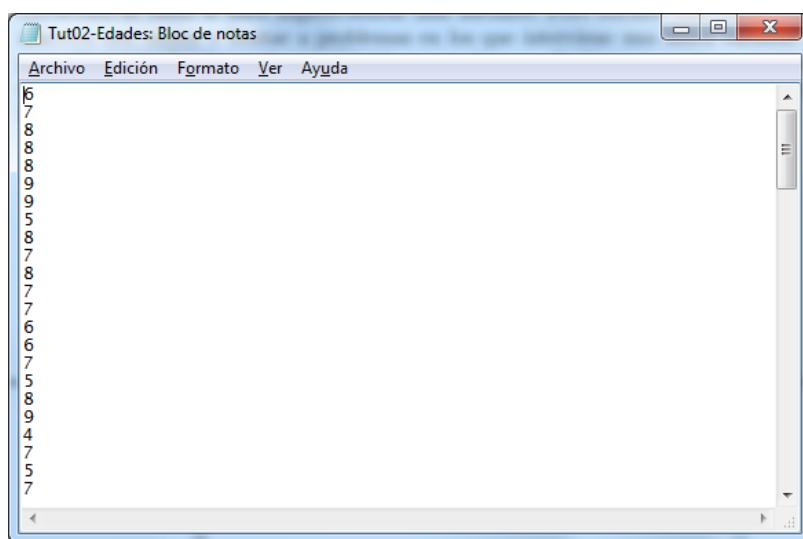
A lo largo del curso iremos comentando otros aspectos relacionados con las buenas prácticas en programación en general y en la documentación del código en particular.

## 7. Ficheros csv en Python.

En esta sección vamos a aprender a utilizar ficheros csv con Python. En el Tutorial-01 hemos visto algunos ejemplos de ese tipo de ficheros y su manejo básico con una hoja de cálculo como Calc. Como vimos allí, un fichero csv típico contiene una tabla de datos, con varias columnas. Esa estructura de tabla se hará imprescindible más adelante. Pero durante una buena parte del curso, nosotros nos vamos a limitar a problemas en los que interviene una única variable. En tal caso, para almacenar los datos de esa variable nos podemos a limitar a considerar un tipo de ficheros csv muy básico, como el fichero adjunto:

[Tut02-Edades.csv](#)

Guárdalo en la subcarpeta **datos** de tu directorio de trabajo (recuerda la estructura de directorios que hemos creado en el Tutorial-00 para que el código del curso funcione sin problemas). Si abres ese fichero con un editor de texto (como el *Bloc de Notas*, en Windows), verás, como muestra esta figura,



que el fichero contiene sólo una columna de datos, que en este ejemplo corresponde a valores de una variable cuantitativa discreta. En la figura sólo se muestra una parte de los datos.

### Ejercicio 19.

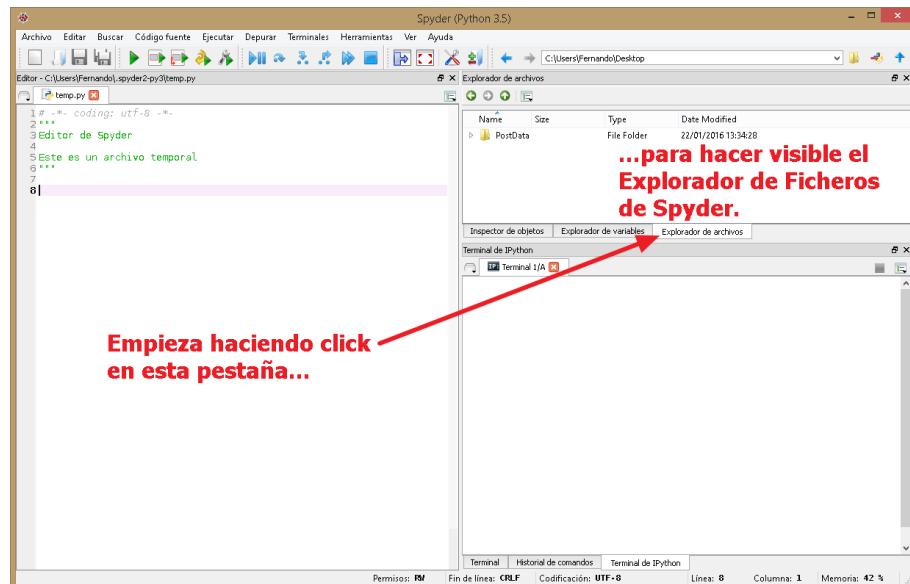
¿Cuántos datos hay en ese vector? Usa Calc para averiguarlo, pronto usaremos la función `len` que vimos antes para hacerlo con Python. □

### Leyendo datos de un fichero csv.

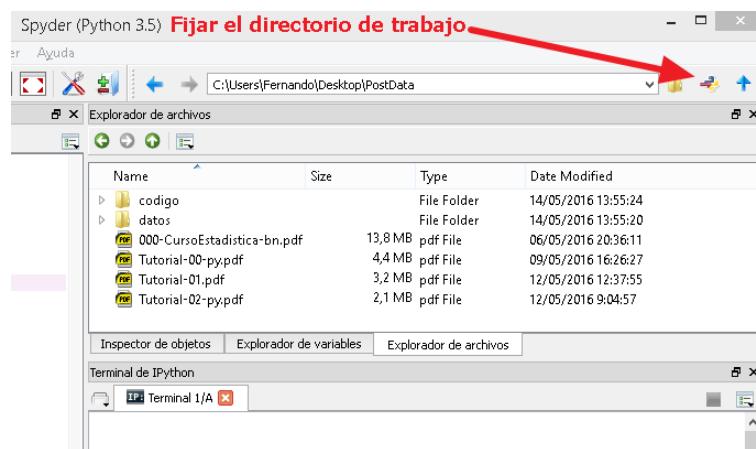
Queremos utilizar los datos de ese fichero en Python, y para eso vamos a tener que dar los siguientes pasos:

1. Indicarle a Python cuál es nuestro *Directorio de Trabajo* para que, tomándolo como referencia, pueda saber llegar hasta el fichero de datos.
2. Leerlos desde el fichero.
3. Y guardarlos en una lista. El resultado será como si nosotros hubiéramos creado ese vector tecleando sus elementos directamente.

Para dar el primer paso vamos a aprender a usar otro componente de Spyder. La siguiente figura ilustra la forma de localizar y activar el panel denominado *Explorador de Archivos* o *File Explorer* si tu versión de Spyder está en inglés.



En ese panel puedes usar el ratón para seleccionar tu *Directorio de Trabajo*. Si necesitas ascender por la estructura de carpetas de tu sistema puedes usar las flechas de color verde que aparecen encima del panel. Navega por las carpetas hasta que el contenido de tu *Directorio de Trabajo* sea visible en el *Explorador de Archivos*. En esta figura se ilustra esa situación:



**¡Cuidado!** No te confundas y selecciones la carpeta **datos** en lugar del *Directorio de Trabajo*.

Como indica la flecha roja, cuando hayas hecho esto, haz click sobre el ícono que aparece encima del panel del *Explorador de Archivos*. Con esto hemos completado la primera de las tres tareas. Pero para comprobar que todo ha ido bien, vamos a ir a la *Terminal* de Spyder y vamos a usar una nueva función mágica (como `%clear` y `%reset`). En este caso debes ejecutar:

```
%ls
```

Al hacerlo verás aparecer en la *Terminal* un listado, parecido al de esta figura, que debe reflejar el contenido de tu *Directorio de Trabajo*.

```

Terminal de IPython
In [1]: %ls
El volumen de la unidad C no tiene etiqueta.
El n mero de serie del volumen es: C06F-BBBC

Directorio de C:\Users\Fernando\Desktop\PostData

14/05/2016 13:55    <DIR>    .
14/05/2016 13:55    <DIR>    ..
06/05/2016 20:36  14.480.147 000-CursoEstadistica.bn.pdf
14/05/2016 13:55    <DIR>    codigo
14/05/2016 13:55    <DIR>    datos
09/05/2016 16:26      4.643.765 Tutorial-00.py.pdf
12/05/2016 12:37      3.335.484 Tutorial-01.pdf
12/05/2016 09:04      2.215.823 Tutorial-02.py.pdf
4 archivos       24.675.219 bytes
4 dirs          54.603.935.744 bytes libres

In [2]:

```

Ahora estamos listos para leer el fichero. El código que vamos a utilizar para esto hace uso de la función `read_csv` que debemos importar desde el módulo `pandas`. A lo largo del curso nos vamos a encontrar en varias ocasiones con este módulo que contiene muchas funciones orientadas a la Estadística y el Análisis de Datos. Los comandos necesarios para las operaciones que hemos descrito aparecen aquí debajo. Ejecutar este código para ver el resultado. Más abajo comentaremos uno a uno los comandos:

```
import pandas as pd
listaEdades = pd.read_csv("./datos/Tut02-Edades.csv", names=["edades"])
listaEdades = listaEdades["edades"].tolist()
print(listaEdades)
```

Vamos a ver paso a paso cómo se ha leído el contenido de ese fichero de datos y se ha convertido en la lista `listaEdades`. No es necesario que en este momento entiendas todos los detalles que vamos a presentar (y la misma observación sirve para el resto de métodos de lectura/escritura que vamos a ver en esta sección). Basta con una comprensión somera del método para que puedas aplicarlo a la lectura de otros ficheros de datos similares. Más adelante, cuando hayas ganado confianza con Python, podrás volver aquí y tratar de entender en detalle cómo funciona la lectura de datos.

- La primera línea simplemente importa el módulo `pandas` con el alias `pd`, que es el que la mayoría de programadores de Python usan habitualmente para este módulo.
- La segunda línea es la más importante: aquí es donde usamos la función `read_csv` para leer los datos del fichero. Pero `pandas` es una librería sofisticada, diseñada para tratar problemas avanzados. Así que la función `read_csv` es capaz de leer ficheros de datos bastante más complejos que el que estamos usando como ejemplo. En particular, el resultado de `read_csv` **no es una lista**, sino un objeto propio de `pandas` llamado `DataFrame` y pensado para almacenar una tabla con varias columnas. En nuestro caso se trata de una tabla con una única columna, pero aún así `pandas` sigue pensando en este objeto como una tabla. En la cuarta línea lo convertiremos en una lista. Pero para eso, por razones técnicas, tenemos que darle un nombre a la columna (única) que contiene los datos. Por eso aparece el argumento opcional `names=["edades"]` en la función `read_csv`.
- Todavía en esa línea, fíjate en cómo hemos indicado la ubicación del fichero, con la cadena de caracteres `./datos/Tut02-Edades.csv`. El punto inicial representa para Python (y para muchos otros programas) su *Directorio de Trabajo*. Así que esta es nuestra forma de decirle a Python (a `pandas` en particular) que vamos a usar concretamente el fichero `Tut02-Edades.csv` situado en la subcarpeta `datos` dentro de su *Directorio de Trabajo*, que ya hemos establecido con anterioridad.
- En la tercera línea de código llevamos a cabo la conversión de la columna del `DataFrame` que hemos llamado `edades` en una lista. Para ello seleccionamos esa columna mediante `listaEdades["edades"]`. Fíjate en que esa forma de seleccionar con corchetes es parecida a la selección de elementos de listas, aunque aquí seleccionamos por nombre y no por posición. Después de seleccionar la columna invocamos el método `tolist` que convierte esa columna en una lista.

En el siguiente ejercicio vas a tener ocasión de practicar la lectura de este tipo de ficheros `csv`.

### Ejercicio 20.

*Guarda el fichero de datos adjunto:*

[\*Tut02-ejercicioLecturaCsv.csv\*](#)

*en tu carpeta `datos`. Ábrelo primero con un editor de texto para hacer una exploración preliminar del fichero. ¡Acostúmbrate a hacer siempre esto! Despues usa el método que hemos visto con la función `read_csv` de `pandas` para leer los datos del fichero con Python y calcula la media aritmética de esos datos.* □

Para cerrar este apartado, es conveniente haber visto el formato del mensaje de error que se produce cuando el fichero que tratamos de leer desde Python no existe, o no está en el directorio de trabajo.

### Ejercicio 21.

Prueba a ejecutar:

```
pd.read_csv("./datos/EsteFicheroNoExiste.csv", names=["v"])
```

y fíjate en el mensaje de error.

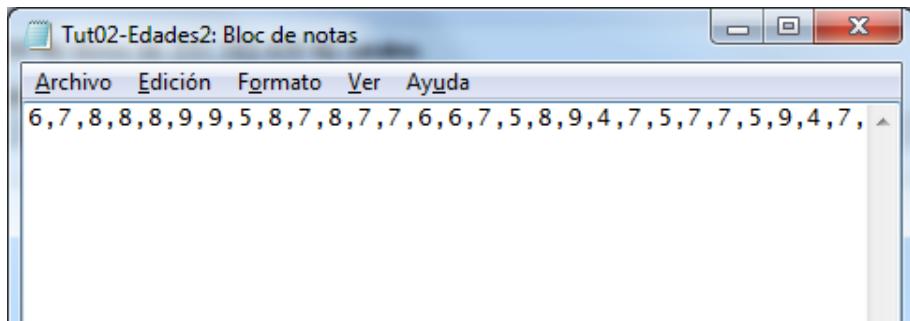


### Otro formato del fichero de datos.

Los ficheros `csv` que contienen un único vector (en lugar de una *tabla* de datos), pueden adoptar formatos distintos del que acabamos de ver. Por ejemplo, el fichero adjunto

[Tut02-Edades2.csv](#)

contiene los mismos datos que `Tut02-Edades.csv`, pero ahora los datos del vector están todos en una fila, y separados por comas. GUÁRDALO, como antes, en la subcarpeta `datos` de tu directorio de trabajo. El aspecto del fichero, visto a través de un editor de texto, es este (sólo son visibles una parte de los datos):



Desde luego, no nos podemos plantear transformar a mano este vector en uno como el del apartado anterior. Podríamos buscar soluciones pasando por la hoja de cálculo (por ejemplo: lo leeríamos en una fila de la hoja, y luego habría que usar el **Pegado especial** para trasponerlo -es decir, girarlo- para finalmente volver a guardarlo). Afortunadamente, hay una solución, sin salir de Python, bastante menos embrollada. Tenemos que usar un argumento opcional de la función `read_csv`, concretamente el argumento `lineterminator` que como su nombre indica sirve para indicarle a `pandas` cuál es el carácter que usamos para separar en líneas nuestro fichero. Ejecuta este código en Spyder para leer el fichero (se asume que ya hemos importado `pandas` y que has fijado el directorio de trabajo):

```
listaEdades2 = pd.read_csv("./datos/Tut02-Edades2.csv", names=["edades"], lineterminator=",")  
  
listaEdades2 = listaEdades2["edades"].tolist()  
  
print(listaEdades2)
```

Como en el apartado anterior, el resultado es una lista de Python que contiene los datos del fichero `csv`.

### Escribiendo datos a un fichero csv.

Para completar nuestra primera visita al manejo de ficheros `csv` desde Python vamos a recorrer el camino inverso. Porque muchas veces, después de hacer operaciones en Python, obtendremos como resultados listas de datos interesantes (y, más adelante en el curso, otro tipo de objetos, como tablas). Lo natural, entonces, es aprender a guardar esos datos en un fichero de tipo `csv`, como el fichero con el que empezamos. De esa forma, por ejemplo, puedes compartir tus resultados con otras personas, incluso aunque no utilicen Python.

Vamos a practicar esto escribiendo a un fichero `csv` la siguiente lista de datos:

```
edades3 = [29, 28, 36, 41, 41, 33, 28, 32, 35, 36, 36, 33, 40, 41, 28, 30, 27, 33, 38, 36]
```

Para hacerlo seguimos un camino inverso al de los apartados anteriores. Primero usamos una función de `pandas` llamada también `DataFrame` que convertirá nuestra lista en un objeto de ese tipo `DataFrame`, que como ya hemos dicho es la estructura de datos básica de `pandas` para representar datos. Recuerda que se supone que hemos importado `pandas` con el alias `pd`:

```
edades3pd = pd.DataFrame(edades3)
```

Hemos añadido `pd` al final del nombre simplemente como recordatorio de que el resultado es un objeto propio de `pandas`. Ahora usamos el método `to_csv` de ese objeto. Recuerda la sintaxis de objetos y métodos que vimos en el apartado 5.1 (pág. 40):

```
edades3pd.to_csv("./datos/Tut02py-Edades-3.csv", header=False, index=False)
```

Los argumentos `header=False` e `index=False` sirven respectivamente para evitar que `pandas` añada al fichero csv una línea de encabezamiento y que numere cada una de las líneas de datos.

### Ejercicio 22.

*Usa un editor de texto para explorar el fichero csv resultante. Comprueba lo que sucede si eliminas uno o ambos argumentos opcionales del método `to_csv`* □

Con esto concluye nuestra breve visita al manejo de ficheros csv desde Python. A lo largo del curso aprenderemos más sobre este ingrediente fundamental para poder comunicar nuestras sesiones de trabajo en Python con el mundo exterior.

## 8. Estadística descriptiva de una variable cuantitativa discreta con datos no agrupados.

Con el trabajo de las secciones previas estamos listos para abordar el objetivo principal de este tutorial: dada una muestra

$$x_1, x_2, \dots, x_n$$

de una variable cuantitativa vamos a describir esa muestra calculando sus medidas centrales o de posición(media, mediana), las medidas de dispersión (varianza, desviación típica) y además las representaciones gráficas que nos ayudan a hacernos una mejor idea de las propiedades de esa muestra. El punto de partida será un fichero csv que contiene los datos de la muestra. Supondremos que en ese fichero los datos están en columna, de manera que hay un único dato en cada fila del fichero. Si no es así, ya hemos visto cómo adaptar el código a otras situaciones frecuentes. En esta sección vamos a usar como ejemplo el fichero adjunto:

[Tut02-var3.csv](#)

### Ejercicio 23.

*Antes de seguir adelante guarda el fichero Tut02-var3.csv en la carpeta datos del Directorio de trabajo y usa un editor de texto para explorar ese fichero csv.* □

Para analizar estos datos vamos a utilizar un fichero de código Python, el primero de nuestros “ficheros plantilla” de estos tutoriales, que calcule de forma automática todas esas medidas descriptivas. Vamos a ir describiendo el contenido de ese fichero, que aparece aquí adjunto:

[Tut02-estadisticaDescriptiva.py](#)

Para que este fichero funcione correctamente es necesario respetar la estructura de directorios que hemos creado en el Tutorial-00. Así que recuerda que debes guardarlo en la subcarpeta llamada `codigo` dentro de tu *Directorio de trabajo*. Abre el fichero en el *Editor de Código* de Spyder para ir recorriéndolo con nosotros a medida que lo comentamos.

Antes de seguir queremos aclarar un detalle técnico. Cuando abras el fichero verás que incluye unas cuantas líneas de comentario especiales que empiezan con `## ----`. La razón por la que hacemos esto es puramente técnica y tiene que ver con la herramienta de documentación que usamos para escribir estos tutoriales, y no es una característica de Python. Recuerda que lo que caracteriza a un comentario en Python es la presencia del símbolo `#`.

### Cabecera del fichero.

Lo primero que verás en el fichero es un bloque inicial de comentarios que sirven para identificar y describir el programa. A continuación aparecen unas instrucciones básicas de uso, que nos recuerdan la necesidad de tener en cuenta cuál es el directorio de trabajo y de la ubicación de los ficheros necesarios respecto de ese directorio.

```

|||||
www.postdata-statistics.com
POSTDATA. Introducción a la Estadística
Tutorial 02.
Plantilla de comandos Python para Estadística Descriptiva
Una variable cuantitativa discreta, datos no agrupados.

|||||
# ATENCIÓN: para que este fichero funcione es NECESARIO:
# (1) Tener en cuenta la estructura de directorios como se explica en el tutorial.
# (2) Introducir el nombre del fichero de datos como argumento de read_csv.

```

## Importando módulos.

El siguiente bloque contiene las líneas de código en las que se importan los módulos que vamos a utilizar.

```

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import collections as cl

```

Las recomendaciones de estilo de Python especifican que todos los módulos deben importarse al comienzo del programa y que cada módulo debe importarse en una línea propia. Fíjate además en que hemos usado alias para los nombres de todos los módulos (de hecho algunos de estos alias son un estándar *de facto* entre los programadores de Python). Ya conoces el módulo `pandas`. Vamos a usar también el módulo `numpy` que contiene muchos objetos y funciones útiles para el Cálculo Numérico. Más adelante en el curso tendremos ocasión de discutir sobre los aspectos numérico y simbólico de las Matemáticas. Por el momento nos conformamos con señalar que `numpy` es uno de los pilares básicos del cálculo científico con Python. Por su parte `matplotlib` es un módulo especializado en gráficas matemáticas, que vamos a usar para dibujar diagramas de barras, de cajas, histogramas, etc. Finalmente, el módulo `collections` aparecerá varias veces en estos tutoriales, pero aquí concretamente lo usaremos para fabricar fácilmente las tablas de frecuencias de nuestros datos.

## Preliminares.

A continuación se incluye un bloque que puede servir para definir algunas variables que se usarán a lo largo del resto del programa.

```

linea = "_" * 75

print(linea)
print(linea)
print("www.postdata-statistics.com")
print("Curso de introducción a la Estadística. Tutorial02 (versión Python).")
print("Estadística descriptiva. Una variable cuantitativa discreta,\n datos no agrupados.")
print(linea)
print(linea)

```

La variable definida mediante `linea = "_"*75` se usa simplemente para dibujar una línea horizontal, y separar así la salida del programa en bloques temáticos. Cada vez que queramos imprimir esa línea separadora usaremos el comando `print(linea)`.

Además, usando `print`, se incluyen en este bloque algunos mensajes que se mostrarán al ejecutar este programa. Siempre es adecuado proporcionar al menos esa información básica al usuario.

## Lectura del fichero de datos. Ejecución del código.

A continuación tenemos el bloque en el que se lee el fichero `csv` que contiene los datos:

```

# Lectura de los datos:
# INTRODUCIR EL NOMBRE DEL FICHERO DE DATOS EN LA SIGUIENTE LINEA:
# EL FICHERO DEBE RESIDIR EN LA CARPETA DATOS DEL DIR. DE TRABAJO
nombreFichero = "Tut02-var3.csv"
datos = pd.read_csv("./datos/" + nombreFichero, names=["v"])
datos = datos["v"].tolist()

print("El fichero de datos es:")
print(nombreFichero)

n = len(datos)
print("El número de datos leídos es:")
print(n)

print("Los primeros 10 datos son:")
print(datos[:10])
print("Los últimos 10 datos son:")
print(datos[-10:])

print(linea)

```

Como ves, para leer los datos se usa la función `read_csv` de `pandas` que ya conocemos. Para un uso correcto del código **es esencial** introducir el nombre del fichero `csv` en la línea adecuada. Después se usa la concatenación (suma) de cadenas de caracteres para obtener como argumento de la función `read_csv` el nombre completo del fichero (incluida la carpeta en la que se encuentra) . Una vez que hemos introducido el nombre del fichero de datos, nos aseguramos de grabar el fichero de código con esa modificación y ya podemos ejecutarlo, como un bloque usando F5 o línea a línea pulsando F9. Si usas la tecla F5 y todo va bien, cuando lo ejecutes verás aparecer en la *Terminal* todos los resultados que produce el programa. En los párrafos que siguen vamos a ir mostrando esos resultados inmediatamente detrás del correspondiente fragmento de código. Si has optado por usar F9 y avanzar línea a línea, los resultados irán apareciendo poco a poco, al ejecutar las líneas que producen salida como texto o gráficos. Por ejemplo, lo primero que hace el código, para comprobar que la lectura de datos ha sido correcta, es mostrar cuántos son los datos leídos (el número se almacena en la variable `n`). Además se muestran los primeros 10 y los últimos 10 valores de la lista de datos. En la *Terminal* eso se traduce en:

```

El número de datos leídos es:
1300
Los primeros 10 valores son:
[4, 8, 4, 4, 5, 5, 3, 6, 6, 2]
Los últimos 10 valores son:
[4, 10, 6, 4, 3, 9, 6, 7, 3, 7]

```

### Recorrido de los datos.

A continuación vamos a determinar el máximo y mínimo de los datos, que conjuntamente determinan lo que hemos llamado el recorrido.

```

## Recorrido de una lista de números.

print("El mínimo y máximo de los datos determinan el recorrido:")
print("Mínimo:")
print(min(datos))

print("Máximo:")
print(max(datos))

print("La anchura del recorrido (max - min) es:")

print(max(datos) - min(datos))

```

```
print(linea)
```

El resultado es:

```
El mínimo y máximo de los datos determinan el recorrido:  
Mínimo:  
0  
Máximo:  
16  
La anchura del recorrido (max - min) es:  
16
```

### Tablas de frecuencia. Tuplas en Python.

Nuestro siguiente objetivo es obtener las tablas de frecuencia de los datos. Primero vamos a construir los valores que deben aparecer en ellas, y después usaremos `print` con formato para mostrar esa información de una manera más cómoda.

Empezamos con la tabla de frecuencias absolutas. Para fabricarla vamos a crear un objeto de tipo `Counter`, procedente del módulo `collections` (importado con el alias `c1`). El código es este, que comentaremos a continuación:

```
datos_counter = c1.Counter(datos)  
tablaFreqAbs = datos_counter.most_common()  
tablaFreqAbs.sort()
```

En la primera línea creamos el objeto de tipo `Counter` a partir de la lista `datos`. Estos objetos sirven en Python para obtener tablas de frecuencia, pero también para manipularlas. Para hacer esas operaciones el objeto `Counter` dispone de una serie de métodos. Aquí vamos a usar el método `most_common`, que devuelve como resultado una representación de la tabla de frecuencias como lista de pares. El resultado se almacena en la variable `tablaFreqAbs`. Aunque nuestro código no lo hace, vamos a ver el resultado después de ejecutar el método `most_common` en la *Terminal*:

```
In [25]: tablaFreqAbs = datos_counter.most_common()
```

```
In [26]: print(tablaFreqAbs)  
[(5, 246), (4, 244), (3, 188), (6, 186), (7, 131), (2, 100), (8, 87), (9, 43),  
(10, 28), (1, 25), (0, 9), (11, 6), (12, 2), (13, 2), (14, 2), (16, 1)]
```

El resultado es una tabla de frecuencias, en forma de lista de pares: cada par contiene como primer elemento uno de los números de la lista `datos` y como segundo elemento la frecuencia absoluta de ese número. Pero hay un problema: los pares aparecen desordenados. Y para una tabla de frecuencia lo natural es ordenar los pares usando los valores de `datos`. Por eso hemos aplicado el método `sort` de ordenación *in situ*, que da como resultado:

```
In [28]: tablaFreqAbs.sort()
```

```
In [29]: print(tablaFreqAbs)  
[(0, 9), (1, 25), (2, 100), (3, 188), (4, 244), (5, 246), (6, 186), (7, 131),  
(8, 87), (9, 43), (10, 28), (11, 6), (12, 2), (13, 2), (14, 2), (16, 1)]
```

Y ahora esta lista sí muestra de forma conveniente la tabla de frecuencias. Más abajo en el código usaremos este resultado para imprimir conjuntamente todas las tablas de frecuencia. Pero no podemos seguir adelante sin comentar algo en lo que tal vez ya hayas reparado. Hemos visto que `tablaFreqAbs` es una lista de *pares*. ¿Qué clase de objeto Python son esos pares? Veámoslo. El primero de esos pares se obtiene así, como es de esperar:

```
In [30]: tablaFreqAbs[0]  
Out[30]: (0, 9)
```

Y para saber de qué tipo es usamos `type`:

```
In [31]: type(tablaFreqAbs[0])
Out[31]: tuple
```

Python nos informa de que es un objeto de tipo `tuple`. En español se suele usar `tupla`. La palabra `tupla` es una generalización de las parejas, tríos, etc., de manera que una tupla es una colección de una cantidad cualquiera de elementos, rodeados por paréntesis. Las tuplas son otra estructura de datos de Python, como las listas y las encontraremos a menudo en estos tutoriales. De momento nos conformamos con saber que existen<sup>3</sup>, que es muy fácil crearlas y que se accede a sus elementos de forma análoga a lo que se hace con las listas. Un ejemplo sencillo:

```
In [1]: unaTupla = (1, 2, 6.4, "Hola")
In [2]: unaTupla[2:4]
Out[2]: (6.4, 'Hola')
```

En las operaciones que sigue nos resultará conveniente disponer de dos listas que contengan por separado los elementos que componen las parejas de `tablaFreqAbs`. Para conseguirlo usamos comprensión de listas dos veces:

```
valoresUnicos = [ item[0] for item in tablaFreqAbs]
freqAbs = [ item[1] for item in tablaFreqAbs]
```

El resultado son estas dos listas que mostramos en la *Terminal* (el programa no las muestra como salida, las hemos obtenido después de ejecutarlo directamente escribiendo sus nombres en la *Terminal*). Aprovecharemos para comprobar que la suma de las frecuencias absolutas es la esperada:

```
In [32]: valoresUnicos
Out[32]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16]

In [33]: freqAbs
Out[33]: [9, 25, 100, 188, 244, 246, 186, 131, 87, 43, 28, 6, 2, 2, 2, 1]

In [34]: sum(freqAbs)
Out[34]: 1300
```

A partir de la lista de frecuencias absolutas es fácil obtener la de frecuencias relativas. Basta con dividir cada frecuencia absoluta por la variable `n`, que almacena el número total de observaciones. Vamos a fabricar esas frecuencias relativas mediante una comprensión de lista:

```
freqRel = [ item/n for item in freqAbs]
```

Como ya hemos anunciado, más abajo usaremos `print` para mostrar todas las frecuencias (absolutas, relativas, etc.) en una misma tabla y con el formato adecuado. Pero podemos mostrar el valor de `freqRel` en la *Terminal* (tras ejecutar el programa):

```
In [35]: print(freqRel)
[0.006923076923076923, 0.019230769230769232, 0.07692307692307693,
0.14461538461538462, 0.18769230769230769, 0.18923076923076923,
0.14307692307692307, 0.10076923076923076, 0.06692307692307692,
0.03307692307692308, 0.021538461538461538, 0.004615384615384616,
0.0015384615384615385, 0.0015384615384615385, 0.0015384615384615385,
0.0007692307692307692]
```

Cuando usemos `print` para mostrar estos valores los redondearemos a una cantidad adecuada de cifras significativas. De momento podemos usar la *Terminal* para comprobar que la suma de las frecuencias relativas es 1, dentro de la precisión que permite el redondeo cuando se trabaja con valores en coma flotante:

<sup>3</sup>Básicamente, existen por razones técnicas, para hacer el código Python más rápido y eficiente. Todo lo que Python hace usando tuplas se podría hacer con listas, pero el código consumiría más recursos de tiempo y memoria.

```
In [36]: sum(freqRel)
Out[36]: 0.9999999999999997
```

Nuestro siguiente objetivo es obtener la tabla de frecuencias acumuladas. Aquí vamos a recurrir por primera vez al módulo `numPy`. Concretamente usaremos la función `cumsum` de ese módulo (el nombre proviene del inglés *cumulative sum*, suma acumulada). Pero el resultado de `cumsum` es un objeto de un tipo que aún no hemos visto, el tipo `ndarray` de `numpy`. Por eso usamos el método `tolist` para convertirlo en una lista.

```
freqAcu = np.cumsum(freqAbs).tolist()
```

Como en los casos anteriores, vamos a usar la *Terminal* para explorar estos objetos. Primero usamos `type` para confirmar qué clase de objeto se obtiene con `np-cumsum` y luego usamos `print` para ver el aspecto de ese objeto (antes de convertirlo en lista):

```
In [37]: type(np.cumsum(freqAbs))
Out[37]: numpy.ndarray

In [38]: print(np.cumsum(freqAbs))
[ 9  34 134 322 566 812 998 1129 1216 1259 1287 1293 1295 1297 1299
1300]
```

Fíjate en que aunque a primera vista pueda parecer una lista, la falta de comas entre los elementos delata que estamos ante otro tipo de objeto. Al aplicar el método `tolist` sí que obtenemos una lista:

```
In [39]: print(np.cumsum(freqAbs).tolist())
[9, 34, 134, 322, 566, 812, 998, 1129, 1216, 1259, 1287, 1293, 1295, 1297, 1299, 1300]
```

y esa lista es la que hemos llamado `freqAcu`. Comprueba algunas de esas frecuencias (¡usando Calc, por ejemplo?) y fíjate en que la última frecuencia acumulada tiene el valor esperado.

Finalmente fabricamos la tabla de frecuencias relativas acumuladas (o acumuladas relativas, tanto da). Ahora esto resulta fácil:

```
freqAcuRel = [ item/n for item in freqAcu]
```

La lista resultante es:

```
In [40]: print(freqAcuRel)
[0.006923076923076923, 0.026153846153846153, 0.10307692307692308,
0.24769230769230768, 0.43538461538461537, 0.6246153846153846,
0.7676923076923077, 0.8684615384615385, 0.9353846153846154,
0.9684615384615385, 0.99, 0.9946153846153846, 0.9961538461538462,
0.9976923076923077, 0.9992307692307693, 1.0]
```

Y comprobamos que el último valor de la lista es 1, como debe ser.

#### Ejercicio 24.

Hemos construido la lista haciendo relativas las frecuencias acumuladas. Haz la cuenta al revés: acumula las frecuencias relativas. Comprueba que obtienes los mismos valores (es posible que veas algunas pequeñas diferencias debidas al redondeo).  $\square$

Ahora que ya hemos obtenido esas cuatro tablas de frecuencias estamos listos para mostrarlas todas en una tabla resumen. Para ello usaremos un bucle `for` y la función `print` con formato, como hemos aprendido a hacer. El código es este, que comentaremos a continuación:

```
k = len(valoresUnicos)
print("\nTablas de frecuencias:\n")
linea = "_" * 75
print(linea)
print("Valor | Frec. absoluta | Frec. relativa | Frec. acumulada | Frec. rel. ac. |")
print(linea)
```

```

for i in range(0,k):
    print("{0:5.3g} | {1:14.3g} | {2:14.3f} |{3:16.3g} |{4:15.3g} |\
        ".format(valoresUnicos[i], freqAbs[i], freqRel[i], freqAcu[i], freqAcuRel[i]))
print(linea)

```

Se trata de un código bastante sencillo. Los parámetros de formato, como el `{3:16.3g}` de la cuarta columna, se han ajustado por ensayo y error tras inspeccionar una ejecución preliminar del código. El resultado al ejecutar el código aparece en la Tabla 2 (pág. 60).

Tablas de frecuencias:

Valor	Frec. absoluta	Frec. relativa	Frec. acumulada	Frec. rel. ac.
0	9	0.007	9	0.00692
1	25	0.019	34	0.0262
2	100	0.077	134	0.103
3	188	0.145	322	0.248
4	244	0.188	566	0.435
5	246	0.189	812	0.625
6	186	0.143	998	0.768
7	131	0.101	1.13e+03	0.868
8	87	0.067	1.22e+03	0.935
9	43	0.033	1.26e+03	0.968
10	28	0.022	1.29e+03	0.99
11	6	0.005	1.29e+03	0.995
12	2	0.002	1.3e+03	0.996
13	2	0.002	1.3e+03	0.998
14	2	0.002	1.3e+03	0.999
16	1	0.001	1.3e+03	1

Tabla 2: Tabla de frecuencias de los datos.

### Ejercicio 25.

A la vista de esta tabla, ¿cuál es tu estimación de la media y la mediana de estos datos? No se espera un cálculo exacto sino una estimación.  $\square$

### Medidas de posición.

Vamos a ocuparnos ahora de las medidas de posición: mediana, cuartiles, percentiles. Para obtenerlas nos vamos a apoyar en el módulo `numpy`, que contiene las funciones `median` y `percentile` para el cálculo de estas cantidades. Al examinar el siguiente fragmento de código fíjate en que podemos calcular varios percentiles a la vez, usando una lista de valores entre 0 y 100 como argumento de la función `percentile`. El resultado de esa función es un `ndarray` de `numpy` (compruébalo usando `type`), como ya vimos que sucedía con `cumsum` al calcular las frecuencias acumuladas. Por eso lo hemos convertido usando `list`.

```

print("Mediana:")
print(np.median(datos))
print("Percentiles 0, 25, 50, 75, 100:")
print(list(np.percentile(datos, [0, 25, 50, 75, 100])))
IQR = np.percentile(datos, 75) - np.percentile(datos, 25)
print("Recorrido intercuartílico:")
print(IQR)

print(linea)

```

El resultado del código anterior es:

```

Mediana (NumPy)
5.0
Percentiles 0, 25, 50, 75, 100 (NumPy)
[0.0, 4.0, 5.0, 6.0, 16.0]
Recorrido intercuartílico
2.0

```

## Gráficos.

El siguiente paso en el código es la representación gráfica de los datos. Python dispone de funciones fáciles de usar para estos gráficos, muchas de las cuales están incluidas en el módulo `matplotlib`. La primera línea de este bloque de código es:

```
get_ipython().magic('matplotlib inline')
```

Esta es la forma de invocar una *función mágica* de IPython desde un archivo de código Python. La función mágica en este caso es `matplotlib inline`. El objetivo es que al ejecutar este fichero las gráficas aparezcan intercaladas en la salida del programa, junto con el resto de valores que producimos usando `print`. Si no usáramos esta función, al ejecutar el código en la *Terminal* cada gráfico se abriría en su propia ventana. Eso tiene algunas ventajas, porque esas ventanas gráficas permiten desplazar el gráfico, hacer zoom y otras manipulaciones que pueden resultar interesantes para explorar gráficas complejas. Pero en estos ejemplos sencillos nos conformamos con algo más simple. Más adelante volveremos sobre este asunto de las ventanas gráficas.

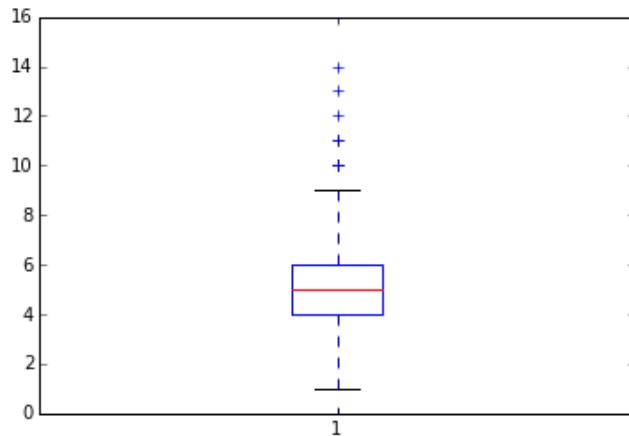
Empecemos a dibujar, pues. Para enlazar con la información que proporcionan las medidas de posición primero dibujaremos un diagrama de cajas (boxplot).

```

print("Diagrama de cajas (boxplot):")
plt.boxplot(datos)
plt.show()

```

El resultado es este gráfico, que puedes comparar con los resultados que obtuvimos para las medidas de posición:



Como vamos a ver, dibujar un gráfico con `matplotlib` es un proceso en dos etapas:

1. Para *construir* el gráfico usamos la función `boxplot` del módulo `matplotlib` (por eso el prefijo `plt`).
2. Para *mostrar* el gráfico usamos la función `show`, también de `matplotlib`.

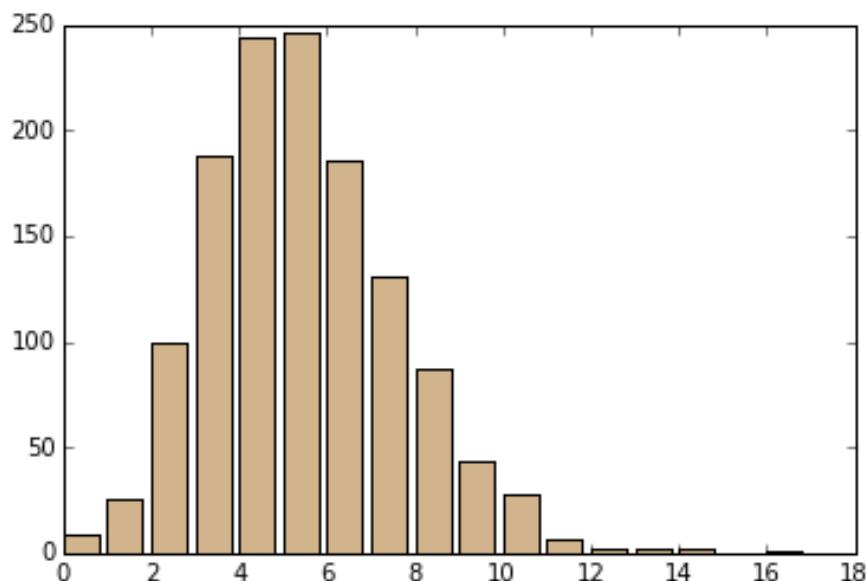
Aunque al principio pueda parecer complicado, esto nos permitirá más adelante utilizar varios comandos para construir gráficos complicados combinando los resultados de esos comandos en una sola gráfica que finalmente mostraremos con `show`.

El siguiente gráfico que vamos a construir es un diagrama de barras (o columnas), que representa gráficamente la información de nuestra tabla de frecuencias. La posición sobre el eje horizontal de cada barra corresponde con uno de los valores de la primera columna de esa tabla (los valores distintos que aparecen en los datos), mientras que la altura de cada una de las barras queda

determinada por la frecuencia absoluta de ese valor. Así que el código empieza identificando esas listas de valores con los nombres `posiciones` y `alturas`. Esto no era, desde luego, necesario, pero ayuda a mejorar la legibilidad del código. Para construir el gráfico usamos la función `bar` de `matplotlib` a la que, aparte de `posiciones` y `alturas`, hemos añadido el argumento opcional `color='tan'` para modificar el color de relleno de las barras del diagrama (el color por defecto es azul oscuro). Como antes, usamos `show` para mostrar el gráfico resultante.

```
print("Diagrama de barras a partir de la tabla de frecuencias:")
posiciones = valoresUnicos
alturas = freqAbs
plt.bar(posiciones, alturas, color='tan')
plt.show()
```

El diagrama de barras que se obtiene es este:



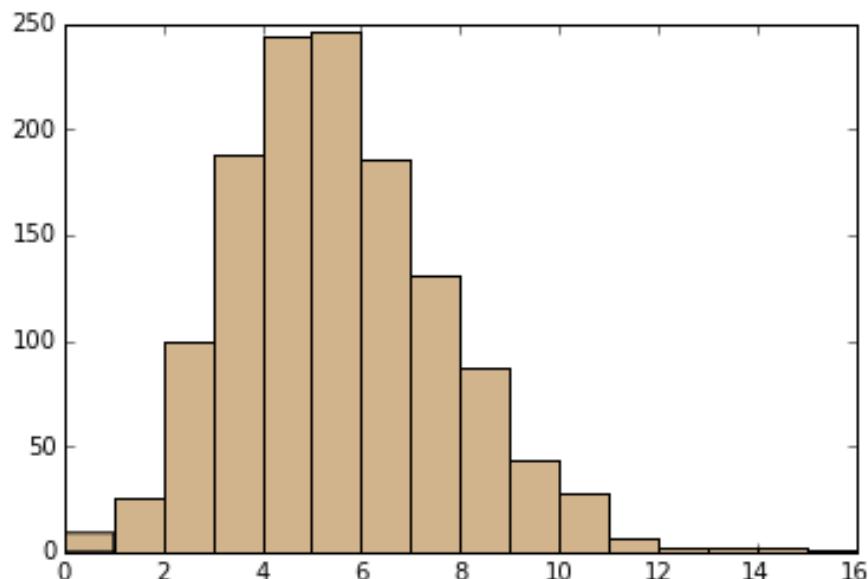
Si te fijas bien, verás que los valores del eje no están correctamente centrados en las columnas del gráfico. Por el momento lo dejamos pasar. Más adelante nos iremos ocupando de este y otros detalles, para mejorar la calidad de los gráficos.

El último gráfico que vamos a construir es un histograma. Puesto que estamos tratando con una variable cuantitativa discreta, el histograma no es nuestra elección prioritaria para representar estos datos: el diagrama de barras es mejor para esta situación. Pero hecha esa advertencia, queremos aprovechar para mostrar la facilidad con la que Python permite construir histogramas. El código es este:

```
print("Histograma:")
plt.hist(datos, bins=len(valoresUnicos), color='tan')
plt.show()

print(linea)
```

Como ves, la función de `matplotlib` responsable de construir el histograma se llama `hist`. El único argumento necesario para `hist` es la lista de datos. Pero podemos usar el argumento opcional `bins` para indicar el número de clases en las que queremos agrupar los datos para representarlos (*bin* en inglés significa *caja*, *bote* o *compartimento*). En este caso hemos hecho que haya tantas cajas como valores distintos para que el perfil del histograma fuera bastante parecido al del diagrama de barras. Y como antes hemos cambiado el color de las barras del gráfico.



Fíjate en las diferencias y similitudes entre el diagrama de barras y el histograma.

### Media aritmética y medidas de dispersión.

En este apartado vamos a usar funciones de `numpy` para calcular rápidamente la media aritmética y varias medidas de dispersión de los datos. Empezando por la media, el cálculo usa la función `mean`

```
print("Media aritmética:")
mediaAritmetica = np.mean(datos)
print(mediaAritmetica)
```

El resultado es:

```
Media aritmética:
5.03923076923
```

El módulo `numpy` incluye dos funciones para calcular las medidas de dispersión, llamadas `var` y `std`. La primera de estas funciones sirve para calcular la varianza poblacional y la cuasivarianza muestral. Para saber cuál de ellas calculamos existe un argumento llamado `ddof` (del inglés *delta degrees of freedom*). Ya sabes que para calcular la cuasivarianza muestral el denominador es  $n - 1$ , mientras que para la varianza poblacional el denominador es  $n = n - 0$  (siendo  $n$  el número de datos). El valor de `ddof` es el valor *que se resta de n*. Por eso para la varianza poblacional usamos `ddof = 0`, mientras que para la cuasivarianza muestral usamos `ddof = 1`.

```
print("Varianza poblacional")
varPoblacional = np.var(datos, ddof=0)
print(varPoblacional)
print("Cuasivarianza muestral")
cuasivarMuestral = np.var(datos, ddof=1)
print(cuasivarMuestral)
```

Con la desviación típica poblacional y la cuasidesviación típica muestral las cosas son muy parecidas, cambiando `var` por `std` (de *standard deviation*):

```
print("Desviación típica poblacional")
desvestPoblacional = np.std(datos, ddof=0)
print(desvestPoblacional)
print("Cuasidesviación típica muestral")
cuasidesvestMuestral = np.std(datos, ddof=1)
print(cuasidesvestMuestral)

print(linea)
```

El resultado de ese código es:

```
Varianza poblacional  
4.71153786982  
Cuasivarianza muestral  
4.71516491976  
Desviación típica poblacional  
2.17060771901  
Cuasidesviación típica muestral  
2.17144305008
```

Vamos a explorar algunas de las ideas que aparecen en ese programa a través de los apartados de este ejercicio.

### Ejercicio 26.

1. Para empezar, si aún no lo has hecho, ejecuta el programa en la Terminal utilizando F5 o F9. Recuerda que debes en primer lugar asegurarte de que tanto el fichero de código py como el fichero de datos csv están situados en las carpetas adecuadas. Además debes incluir el nombre del fichero de datos en el fichero de código Python para que la función `read_csv` pueda hacer su trabajo. Tras ejecutar el código y ver aparecer los resultados conseguirás que todas las variables que se definen en el código sean accesibles en la Terminal. En particular la variable `datos` contiene la lista de datos procedente del fichero csv.
2. Calcula el percentil 20 de los datos.
3. Comprueba que el valor de la media aritmética que produce el programa (y que está almacenado en la variable `mediaAritmetica`) coincide con lo que obtienes dividiendo la suma de datos por `n`.
4. Usa código como el del fichero `Tut02-mediaVarianza-01.py` (ver pág. ??) para calcular la varianza y la desviación típica poblacionales y comprueba que coinciden con los resultados de numpy (puede haber una pequeña diferencia debida al redondeo). Haz lo mismo con la cuasivarianza y la cuasidesviación típica muestrales.
5. Ejecuta de nuevo el programa con F5, pero antes de hacerlo comenta la línea

```
get_ipython().magic('matplotlib inline')
```

Es decir, cámbiala a:

```
# get_ipython().magic('matplotlib inline')
```

Acuérdate de grabar el programa con ese cambio antes de volver a ejecutarlo (y de deshacer el cambio cuando acabes este ejercicio). ¿Qué sucede ahora al ejecutar el programa? Si todo va como se espera verás aparecer una ventana gráfica con el diagrama de cajas. Para seguir avanzando debes cerrar esta ventana. Cuando lo hagas aparecerá otra ventana con el siguiente gráfico, el de barras. Al cerrar esta aparecerá el histograma y finalmente, al cerrar esta concluirá la ejecución del resto del programa. Este es el comportamiento típico de IPython cuando no se usa la orden `get_ipython().magic('matplotlib inline')`. En programas que muestran un gran número de gráficos eso puede resultar molesto, así que es bueno que conozcas las opciones de las que dispones.

6. Dibuja un gráfico de barras en el que la altura de las barras corresponda a las frecuencias acumuladas en lugar de las absolutas. Más adelante en el curso volveremos sobre las diferencias entre estos dos tipos de gráficos.



## 9. Más operaciones con listas.

### 9.1. Números aleatorios.

En el Tutorial-01 vimos como generar números (pseudo)aleatorios con Calc. Y en el Capítulo 3 de la teoría del curso se usan esos números para hacer varios experimentos relacionados con las probabilidades, en situaciones bastante elementales. Para prepararnos, vamos a aprender a hacer lo mismo con Python. Como veremos, vamos a poder ir mucho más allá de lo que resulta viable hacer con Calc.

Muchas de las funciones que vamos a utilizar se ubican en el módulo `random`. Así que empezaremos importándolo.

```
import random as rnd
```

La primera de las funciones de este módulo que vamos a examinar es la función `randrange`, que nos permitirá simular situaciones sencillas como el lanzamiento de un dado. Vamos a verla en acción en la *Terminal*. Ten en cuenta que cuando tú ejecutes este código obtendrás valores distintos de los que aparecen aquí:

```
In [1]: import random as rnd  
  
In [2]: rnd.randrange(1, 7)  
Out[2]: 2  
  
In [3]: rnd.randrange(1, 7)  
Out[3]: 5  
  
In [4]: rnd.randrange(1, 7)  
Out[4]: 5  
  
In [5]: rnd.randrange(1, 7)  
Out[5]: 6  
  
In [6]: rnd.randrange(1, 7)  
Out[6]: 3  
  
In [7]: rnd.randrange(1, 7)  
Out[7]: 1
```

Como ves, cada vez que ejecutamos la función se obtiene un número entero del 1 al 6. Como en otros casos que hemos visto en Python, usamos `(1, 7)` pero el último número entero de ese intervalo (el 7) se excluye. En general, al escribir `rnd.randrange(a, b)` obtenemos números entre `a` y `b-1`. Por ejemplo, si en lugar de lanzar un dado queremos sacar cartas (sin remplazamiento) de una baraja de 48 cartas, entonces podemos representar las cartas de la baraja con los números del 1 al 48 y bastaría con usar `randrange` así:

```
In [1]: import random as rnd  
  
In [2]: rnd.randrange(1, 49)  
Out[2]: 27  
  
In [3]: rnd.randrange(1, 49)  
Out[3]: 13  
  
In [4]: rnd.randrange(1, 49)  
Out[4]: 36  
  
In [5]: rnd.randrange(1, 49)  
Out[5]: 42  
  
In [6]: rnd.randrange(1, 49)  
Out[6]: 6
```

Aunque podemos usar la función así para fabricar unos pocos valores, lo que de verdad vamos a necesitar es la capacidad de hacer simulaciones en las que un experimento se repita cientos o miles de veces. Vamos a combinar la función `randrange` con la comprensión de listas para simular 100 tiradas de un dado:

```
In [1]: import random as rnd

In [2]: dado100 = [rnd.randrange(1, 7) for _ in range(0, 100)]

In [3]: print(dado100)
[5, 5, 4, 6, 6, 4, 4, 6, 5, 1, 6, 6, 3, 2, 4, 6, 5, 6, 2, 5, 1, 2, 6, 2, 6,
4, 3, 2, 5, 1, 1, 5, 1, 3, 3, 6, 5, 6, 6, 5, 6, 1, 6, 3, 4, 1, 6, 1, 4, 2, 5, 2, 5,
1, 3, 4, 3, 4, 6, 5, 2, 4, 2, 5, 3, 2, 3, 4, 1, 1, 3, 4, 6, 3, 4, 1, 6, 3, 5, 2, 2,
1, 3, 3, 2, 2, 5, 2, 4, 3, 2, 2, 3, 5, 6, 5, 3]
```

Un detalle técnico: fíjate en que hemos escrito `for _ in range(0, 100)`, usando un guión bajo `_` para representar la variable auxiliar del `for`. También podríamos haber escrito `for item in range(0, 100)`, usando una variable auxiliar `item`, y el resultado habría sido el mismo. Pero los programadores de Python usan a menudo ese convenio de notación llamando `_` a la variable auxiliar *cuando esa variable no se utiliza en ningún otro punto del código y es simplemente un contador de iteraciones*.

Hasta ahora, hemos visto ejemplos en los que extraímos valores aleatorios de intervalos que empiezan en 1, como `1:6`, y `1:48`. Naturalmente, podemos aplicar `randrange` a un intervalo como `161:234`. Otras veces, en cambio, nos sucederá que tenemos una lista, como esta lista `edades`:

```
edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19]
```

y lo que queremos es extraer algunos de estos valores al azar; pongamos por ejemplo, que queremos extraer 7 elementos. Para hacer eso, si queremos muestreo sin remplazamiento podemos usar la función `sample` del módulo `random`. Veamos como:

```
In [1]: import random as rnd

In [2]: edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19]

In [3]: edadesAzar_NoRemp = rnd.sample(edades, 7)

In [4]: print(edadesAzar_NoRemp)
[18, 21, 17, 20, 18, 20, 17]
```

En cambio, si lo que queremos es extraer elementos con remplazamiento entonces podemos usar la función `choice`. Esta función permite extraer un único elemento al azar de una lista. Para extraer más de uno la combinamos con la comprensión de listas, de la misma forma que hicimos para las tiradas del dado. Por ejemplo, para extraer 100 edades al azar de la lista dada hacemos esto (necesariamente con remplazamiento, claro):

```
In [5]: edadesAzar_Remp = [rnd.choice(edades) for _ in range(0, 100)]

In [6]: print(edadesAzar_Remp)
[17, 20, 19, 17, 20, 18, 19, 19, 19, 17, 18, 18, 19, 18, 20, 19, 17, 22, 19, 17,
20, 18, 22, 18, 21, 20, 18, 22, 20, 22, 22, 19, 18, 20, 20, 19, 21, 21, 22, 22,
18, 19, 20, 20, 21, 18, 22, 17, 22, 19, 20, 17, 18, 21, 18, 17, 21, 21, 22, 22,
22, 20, 22, 17, 21, 19, 17, 18, 22, 18, 21, 19, 19, 22, 22, 18, 22, 19,
21, 19, 19, 18, 18, 20, 19, 19, 21, 22, 17, 22, 18, 19, 18, 20, 22, 18, 18]
```

Fíjate en que, en este caso, la lista original ya contiene elementos repetidos. Así que, independientemente de que el muestreo sea con remplazamiento o sin él, siempre podemos obtener valores repetidos. Tanto `sample` como `choice` eligen al azar *posiciones* dentro de la lista `edades`, y no los *valores* que ocupan esas posiciones. Para entender esto un poco mejor, mira lo que sucede al ejecutar este código, en el que usamos `choice` para extraer 100 valores aleatorios de una lista que tiene el número 1 repetido 9 veces y un único 2:

```
In [7]: muchosUnos = [1,1,1,1,1,1,1,1,1,2]

In [8]: muestra = [rnd.choice(muchosUnos) for _ in range(0, 100)]

In [9]: print(muestra)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Hay otra función del módulo `random` que también usaremos a menudo, la función `shuffle`, que en inglés significa *barajar*. Y como su nombre indica, lo que esta función hace es barajar o reordenar en orden aleatorio los elementos de una lista. Por ejemplo:

```
In [10]: lista = [1, 2, 3, 4, 5, 6, 7, 8]

In [11]: rnd.shuffle(lista)

In [12]: lista
Out[12]: [5, 3, 6, 2, 7, 1, 8, 4]
```

Fíjate en que la reordenación es *in situ*. Si no quieres modificar la lista original, puedes usar `sample` así:

```
In [13]: lista = [1, 2, 3, 4, 5, 6, 7, 8]

In [14]: rnd.sample(lista, len(lista))
Out[14]: [1, 5, 4, 8, 6, 7, 2, 3]

In [15]: lista
Out[15]: [1, 2, 3, 4, 5, 6, 7, 8]
```

Como ves, una muestra obtenida con `sample` de longitud igual a la de la lista original es simplemente una reordenación aleatoria de la lista. Pero este método no afecta a la lista original.

Estas tres funciones `sample`, `choice`, `shuffle` se aplican de la misma forma a listas de cadenas de caracteres, como ilustran estos ejemplos:

```
In [22]: continentes = ["América", "Asia", "Europa", "África", "Oceanía", "Antártida"]

In [23]: rnd.sample(continentes, 3)
Out[23]: ['América', 'Oceanía', 'Europa']

In [24]: muestra = [rnd.choice(continentes) for _ in range(0, 20)]

In [25]: print(muestra)
['Europa', 'África', 'Europa', 'Antártida', 'América', 'Europa', 'Oceanía', 'Asia',
'Europa', 'América', 'Asia', 'América', 'Europa', 'Europa', 'África', 'Oceanía',
'Oceanía', 'América', 'África', 'África']

In [26]: rnd.shuffle(continentes)

In [27]: print(continentes)
['Antártida', 'Europa', 'África', 'Oceanía', 'Asia', 'América']
```

## Usando numpy para fabricar números aleatorios.

Aunque las funciones del módulo `random` son, por el momento, suficientes para nuestras necesidades, es conveniente que conozcas algunas posibilidades que ofrece `numpy`, entre otras razones porque puedes encontrártelas en el código de otras personas. A lo largo del curso iremos presentando cada vez más de estas herramientas. Para empezar podemos usar la función `random.choice` de `numpy`, que ilustra este ejemplo:

```
In [16]: import numpy as np

In [17]: edades = [22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19,
18, 19, 18, 22, 20, 19]

In [18]: np.random.choice(edades, size=100, replace=True)
Out[18]:
array([22, 19, 19, 22, 18, 18, 20, 18, 18, 21, 18, 19, 17, 20, 20, 22, 22,
22, 17, 22, 18, 18, 18, 19, 18, 20, 18, 22, 19, 17, 22, 20, 21, 18,
17, 17, 22, 18, 18, 19, 21, 18, 18, 17, 19, 19, 19, 17, 18, 17,
19, 17, 19, 20, 18, 22, 19, 21, 21, 17, 22, 18, 19, 21, 19, 18,
19, 21, 17, 21, 18, 19, 17, 21, 20, 20, 22, 22, 20, 22, 19, 17,
18, 20, 19, 18, 20, 22, 22, 21, 21, 19, 20, 20])
```

El resultado es similar a lo que obteníamos antes combinando la función `choice` del módulo `random` con la comprensión de listas. Se obtiene una muestra aleatoria con remplazamiento de los elementos de la lista original. Hemos dejado la salida tal cual para que puedas observar que el resultado no es una lista, sino un objeto `ndarray` de `numpy`.

## 9.2. Conjuntos.

Aunque las listas han tenido en este tutorial un papel protagonista, ya hemos anunciado que a lo largo del curso irán apareciendo otras estructuras de datos que resultan necesarias para facilitar nuestro trabajo. En particular, al estudiar la probabilidad necesitaremos trabajar con **conjuntos**. La mayor diferencia entre un conjunto y una lista es que el conjunto no puede contener elementos repetidos. Para definir un conjunto podemos enumerar sus elementos entre llaves. Usamos la *Terminal* definimos y mostramos un conjunto `A`:

```
In [1]: A = {4, -3, 1, 2, 5, 4, 6, 7, 1, 2}

In [2]: print(A)
{1, 2, 4, 5, 6, 7, -3}
```

Fíjate en que Python ha eliminado automáticamente los elementos repetidos de `A`. De paso los ha reordenado de alguna manera extraña. Esa es la otra propiedad importante de los conjuntos de Python: el orden no es importante y a menudo resulta difícil o imposible predecir el orden en que Python colocará los elementos en un conjunto. Si tratamos de acceder a ellos como en una lista sucede esto:

```
In [3]: A[0:4]
Traceback (most recent call last):

File "<ipython-input-3-8c229c82880a>", line 1, in <module>
  A[0:4]

TypeError: 'set' object is not subscriptable
```

Como ves Python nos recuerda *amablemente* que un conjunto no es como una lista. Pero eso no significa que no podamos hacer operaciones sobre los elementos del conjunto, de forma similar a lo que hacíamos en las comprensiones de lista. Por ejemplo, podemos elevar todos los elementos al cuadrado:

```
In [4]: B = {item**2 for item in A}

In [5]: print(B)
{1, 4, 36, 9, 16, 49, 25}
```

Por cierto, aquí tienes una nueva oportunidad de ver lo que decíamos sobre lo difícil de predecir el orden de los elementos.

Al igual que hemos hecho una **comprensión de conjuntos**, podemos usar los elementos de un conjunto en un bucle `for`, como hacemos aquí:

```

for item in A:
    print("El siguiente elemento de A es {} y su cuadrado es {}".format(item, item**2))

```

Al ejecutar este código en la *Terminal* se obtiene:

```

El siguiente elemento de A es 1 y su cuadrado es 1
El siguiente elemento de A es 2 y su cuadrado es 4
El siguiente elemento de A es 4 y su cuadrado es 16
El siguiente elemento de A es 5 y su cuadrado es 25
El siguiente elemento de A es 6 y su cuadrado es 36
El siguiente elemento de A es 7 y su cuadrado es 49
El siguiente elemento de A es -3 y su cuadrado es 9

```

La propiedad de no contener repeticiones es de hecho la primera utilidad que encontramos de la idea de conjunto. Si tenemos una lista de valores y queremos saber cuántos y cuáles son los valores distintos que aparecen en esa lista, podemos convertir la lista en un conjunto. Eso es lo que se hace en el código de este ejemplo:

```

import random as rnd
datos = [rnd.randrange(0, 100) for _ in range(0, 60)]
print("Los 60 datos son: ")
print(datos)
valoresUnicos = set(datos)
print("Entre los datos hay {} valores únicos que son:".format(len(valoresUnicos)))
print(valoresUnicos)

```

El resultado al ejecutarlo es:

```

Los 60 datos son:
[47, 31, 99, 60, 28, 86, 86, 41, 56, 18, 13, 45, 66, 11, 24, 98, 60, 54, 94, 3,
 90, 64, 21, 87, 91, 22, 85, 12, 49, 90, 33, 8, 46, 38, 28, 26, 17, 83, 86, 17,
 71, 0, 66, 96, 2, 22, 33, 49, 50, 2, 64, 3, 47, 62, 82, 80, 17, 72, 0, 80]
Entre los datos hay 43 valores únicos que son:
{0, 2, 3, 8, 11, 12, 13, 17, 18, 21, 22, 24, 26, 28, 31, 33, 38, 41, 45, 46, 47, 49,
 50, 54, 56, 60, 62, 64, 66, 71, 72, 80, 82, 83, 85, 86, 87, 90, 91, 94, 96, 98, 99}

```

Fíjate especialmente en estos dos detalles:

1. La línea:

```
datos = [rnd.randrange(0, 100) for _ in range(0, 60)]
```

sirve para generar 60 números al azar entre 0 y 99. Es un truco que repetiremos más veces en el curso, así que asegúrate de que entiendes como funciona.

2. La función `set` convierte una lista en un conjunto y es, de hecho, la que elimina las repeticiones.

De la misma forma, la función `list` permite convertir un conjunto en una lista y de esa forma recuperar la posibilidad de ordenar los elementos, seleccionar parte de ellos, etc. Por ejemplo, al ejecutar:

```

unicos = list(valoresUnicos)
print(unicos)
print(unicos[4:12])

```

se obtiene:

```

[0, 1, 2, 3, 6, 12, 18, 20, 26, 27, 29, 31, 33, 34, 36, 38, 45, 47, 48, 49, 52, 55,
 56, 57, 59, 61, 64, 65, 66, 67, 74, 76, 77, 78, 80, 83, 84, 88, 92, 95, 97, 98]
[6, 12, 18, 20, 26, 27, 29, 31]

```

Nos hemos asomado apenas a las posibilidades que ofrece el trabajo con conjuntos en Python. Tendremos ocasión más adelante de ver cómo se pueden realizar muchas de las operaciones típicas: uniones, intersecciones, diferencias de conjuntos, etc.

### 9.3. Bucles for anidados.

A veces nos encontramos con esta situación: tenemos una tabla de frecuencias absolutas de un conjunto de datos, pero no tenemos la lista original con los datos. No es una situación infrecuente: a veces al leer un artículo los datos aparecen resumidos en forma de tabla de frecuencias. Y sin embargo, muchas de las operaciones que vamos a realizar asumen que el punto de partida es una lista de datos, repetidos tantas veces como corresponda a su frecuencia. Si antes aprendimos a fabricar la tabla de frecuencias absolutas de una lista de datos, lo que necesitamos ahora es la operación inversa: pasar de la tabla de frecuencias a la lista de datos. Es importante darse cuenta de que en realidad la tabla de frecuencias, por si misma, no permite recuperar por completo la lista original: el orden de los elementos se habrá perdido. Pero no es menos cierto que el orden no juega ningún papel en muchas operaciones estadísticas, como el cálculo de medias, medianas, varianzas, etc.

Para empezar vamos a suponer que los distintos valores y sus frecuencias están disponibles en dos listas:

```
valores = [2,3,5,8,13]
frecuencias = [5.7,12.2,14]
```

Así pues, por ejemplo el valor 5 aparece 12 en los datos originales. Para reconstruir la lista de datos con repeticiones podemos aplicar esta receta en pseudocódigo:

Crear una lista datos, inicialmente vacía.  
Repetir esto para cada posición pos de la lista de valores:  
    Repetir esto un número de veces igual a frecuencias[pos]:  
        Anadir valores[pos] a la lista datos.

Y la traducción en código es esta:

```
datos = []
for i in range(0, len(valores)):
    for _ in range(0, frecuencias[i]):
        datos.append(valores[i])
print(datos)
```

Como ves, tenemos dos bucles `for` anidados: hay que hacer algo para cada posición de la lista `valor`, así que en el primer bucle o **bucle exterior** usamos un contador `i` para recorrer la lista de valores. Pero es que además lo que hay que hacer para cada posición también es un bucle: tenemos que repetir ese valor las veces que nos indica la frecuencia correspondiente. Por eso hay un segundo bucle, o **bucle interno** en el que hacemos algo una cierta cantidad de veces. El contador de este bucle no se usa para ninguna otra operación aparte de llevar la cuenta del número veces: por eso lo llamamos simplemente `_`. Fíjate además en que el número de iteraciones del bucle interno no es constante: depende del valor de `i`; es decir, depende de en qué iteración del bucle exterior nos encontramos.

Veamos cómo funciona esto en la *Terminal* (hemos ejecutado los bucles `for` anidados como un bloque de código, seleccionándolos en el *Editor de Código* y pulsando `F9`):

```
In [1]: valores = [2,3,5,8,13]
In [2]: frecuencias = [5,7,12,2,14]
In [3]: datos = []
In [4]: for i in range(0, len(valores)):
...:     for _ in range(0, frecuencias[i]):
...:         datos.append(valores[i])
...:
In [5]: print(datos)
[2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 8, 8, 13,
13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13]
```

El resultado, como ves, es el deseado: la lista expandida de datos correspondiente a la tabla de frecuencias.

Tal vez te preguntes: ¿se puede hacer lo mismo con una comprensión de listas? Y la respuesta es afirmativa:

Como ves, el orden de escritura de izquierda a derecha de los bucles en esta versión se corresponde con el orden de arriba hacia abajo en los bucles `for` anidados. En cualquier caso, el resultado es el mismo. A menudo la elección entre las dos formas de proceder es una cuestión de preferencias personales. Muchos programadores opinan que los bucles `for` anidados son en general más legibles, pero otros muchos sostienen que la comprensión anidada de listas es la opción *más pythonica*. No vamos a discutir por esto, desde luego. Recuerda en cualquier caso la directriz general que hemos visto anteriormente: las comprensiones pueden ser la forma natural de *fabricar objetos* (como en el caso que nos ocupa), mientras que los bucles `for` son la forma natural de describir *acciones*.

#### 9.4. Listas de cadenas de texto.

Vamos a comentar muy brevemente algunas funciones del módulo **string** que nos resultarán útiles en algunos ejemplos y ejercicios. Este módulo contiene varios objetos que son simplemente cadenas de caracteres con las letras del alfabeto (las que usa el idioma inglés, no incluye acentos ni la ñ) en mayúsculas, minúsculas o ambas conjuntamente:

```
import string
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.ascii_letters)

## abcdefghijklmnopqrstuvwxyz
## ABCDEFGHIJKLMNOPQRSTUVWXYZ
## abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Como decíamos estos objetos resultan interesantes en algunos ejemplos, cuando se combinan con las funciones aleatorias que hemos visto antes. Te proponemos un ejercicio para practicar esto:

### Ejercicio 27.

1. Vamos a fabricar un generador de contraseñas aleatorias. Usa las cadenas de caracteres para escribir un programa que genere contraseñas aleatorias de longitud 20 formadas por letras mayúsculas, minúsculas y dígitos del 0 al 9.
  2. ¿Puedes modificar ese programa para garantizar que las contraseñas contienen al menos: una mayúscula, una minúscula y un dígito?

1

### 9.5. Números pseudoaleatorios, pero “reproducibles”: la función `seed`.

En la Sección 9.1 (pág. 65) hemos lanzado 100 veces un dado, ejecutando este código en la *Terminal* (recuerda que hemos importado `random` con el alias `rnd`):

```
dado100 = [rnd.randrange(1, 7) for _ in range(0, 100)]
```

Un inconveniente de trabajar con números aleatorios es que los resultados del lector serán diferentes de los nuestros y, de hecho, serán diferentes cada vez que ejecutes la función `randrange`. Los números aleatorios se utilizan mucho, por ejemplo, para hacer simulaciones. Y si queremos hacer una de esas simulaciones, y compartirla con otras personas, de manera que puedan *verificar* nuestros resultados, entonces necesitamos:

- Que los números sean aleatorios, en el sentido de que nosotros no los hemos elegido, sino que son el resultado de un *sorteo*.
- Pero que los resultados del sorteo queden registrados de alguna manera, para que otros puedan reproducirlos.

Afortunadamente (en este caso), como ya dijimos, los números que produce un ordenador no son aleatorios, sino pseudoaleatorios. Y para lo que aquí nos ocupa, eso es una ventaja. Hay una función del módulo `random`, llamada `seed`, que permite decirle a Python que queremos hacer exactamente esto: generar números aleatorios reproducibles. Concretamente, para ver funciona como esto, probemos a ejecutar varias veces el código anterior:

```
In [1]: import random as rnd

In [2]: dado100 = [rnd.randrange(1, 7) for _ in range(0, 100)]

In [3]: print(dado100)
[2, 6, 3, 4, 1, 1, 5, 6, 6, 1, 2, 1, 3, 6, 5, 1, 2, 6, 1, 1, 4, 1, 1, 1, 1, 3, 2,
2, 2, 5, 5, 1, 3, 1, 2, 5, 6, 2, 1, 5, 2, 4, 3, 1, 3, 5, 5, 5, 2, 5, 1, 1, 3, 1,
5, 2, 3, 3, 5, 4, 4, 1, 2, 2, 4, 3, 3, 3, 4, 5, 2, 2, 1, 4, 4, 2, 1, 1, 4, 2, 3,
2, 6, 4, 6, 6, 5, 4, 5, 3, 3, 2, 3, 2, 6, 5, 3, 2, 3, 5]

In [4]: dado100 = [rnd.randrange(1, 7) for _ in range(0, 100)]

In [5]: print(dado100)
[6, 1, 3, 4, 3, 3, 4, 4, 1, 6, 1, 4, 4, 1, 5, 6, 4, 1, 5, 2, 4, 6, 3, 1, 1, 5, 4,
1, 4, 6, 2, 1, 2, 5, 2, 1, 2, 4, 1, 3, 2, 6, 1, 6, 5, 6, 2, 5, 4, 3, 1, 4, 1, 3,
3, 6, 5, 6, 2, 2, 5, 4, 2, 1, 6, 1, 4, 2, 3, 5, 3, 2, 1, 3, 4, 2, 6, 3, 2, 6, 1,
6, 5, 2, 3, 3, 3, 1, 4, 5, 4, 6, 6, 3, 6, 4, 5, 5, 2]
```

Como esperábamos, cada vez se obtiene una lista distinta. Vamos a repetir esto, pero ahora ejecutaremos la función `seed` cada vez, antes de llamar a `randrange`:

```
In [6]: rnd.seed(2016)

In [7]: dado100 = [rnd.randrange(1, 7) for _ in range(0, 100)]

In [8]: print(dado100)
[6, 4, 5, 3, 6, 6, 1, 3, 2, 1, 4, 4, 1, 3, 2, 2, 3, 2, 3, 2, 2, 4, 4, 6, 3, 1,
6, 6, 5, 2, 3, 5, 6, 1, 3, 5, 2, 4, 3, 1, 6, 4, 1, 3, 4, 1, 2, 5, 5, 1, 4, 5, 2,
3, 6, 2, 3, 1, 4, 5, 1, 3, 4, 6, 6, 3, 6, 3, 2, 6, 3, 6, 1, 5, 6, 5, 1, 5, 6, 6,
6, 4, 5, 2, 3, 2, 5, 4, 5, 5, 1, 1, 5, 2, 3, 4, 5, 5]

In [9]: rnd.seed(2016)

In [10]: dado100 = [rnd.randrange(1, 7) for _ in range(0, 100)]

In [11]: print(dado100)
[6, 4, 5, 3, 6, 6, 1, 3, 2, 1, 4, 4, 1, 3, 2, 2, 3, 2, 3, 2, 2, 2, 4, 4, 6, 3, 1,
6, 6, 5, 2, 3, 5, 6, 1, 3, 5, 2, 4, 3, 1, 6, 4, 1, 3, 4, 1, 2, 5, 5, 1, 4, 5, 2,
3, 6, 2, 3, 1, 4, 5, 1, 3, 4, 6, 6, 3, 6, 3, 2, 6, 3, 6, 1, 5, 6, 5, 1, 5, 6, 6,
6, 4, 5, 2, 3, 2, 5, 4, 5, 5, 1, 1, 5, 2, 3, 4, 5, 5]
```

Fíjate en que las dos listas `dado100` que hemos obtenido ahora son idénticas. Y si lo ejecutas en tu ordenador, tú también obtendrás exactamente esa misma lista. Eso tiene la ventaja, como decíamos, de que podemos hacer experimentos “al azar”, pero reproducibles por cualquiera que disponga del código.

Como ves, la función `seed` utiliza un argumento, al que llamamos la *semilla* (en inglés, *seed*), que en este caso yo he fijado, arbitrariamente, en 2016. La idea es que si utilizas `seed` con la misma semilla que yo, obtendrás los mismos números pseudoaleatorios que yo he obtenido.

Una vez visto lo fundamental, no queremos entretenernos mucho más en esto. Pero no podemos dejar de mencionar que el asunto de cómo se elige la semilla es delicado. Podría parecer que lo

mejor, en una simulación, es elegir la propia semilla “al azar”. El problema es que, de esa manera, en caso de que alguien sospeche que se han manipulado los datos, puede pensar que hemos ido probando varias de estas semillas “al azar”, hasta obtener unos resultados especialmente buenos de la simulación. Muchos autores recomiendan, como alternativa, fijar una política con respecto a la elección de la semilla, y atenerse a ella en todas las simulaciones. Por ejemplo, puedes usar siempre como semilla el año en que realizas la simulación, como hemos hecho aquí.

### Ejercicio 28.

Ahora que ya hemos aprendido a usar listas aleatorias reproducibles podemos empezar a hacer ejercicios como este.

1. Obtén la tabla de frecuencias (absoluta, relativa, etc.) y calcula la media aritmética y la cuasi-desviación típica (muestral) de los valores que hay en la lista `dado100`. Usa `rnd.seed(2016)` para generar la lista.
2. Para ir preparando el terreno: si eliges un número al azar de esa lista ¿cuál crees que es la probabilidad de que sea un 4? ¿Cuál es la frecuencia relativa de 4 en esa lista?

□

## 10. Instrucción if y valores booleanos.

Para que nuestros programas puedan empezar a ser interesantes les falta aún un ingrediente esencial. Tienen que ser capaces de tomar decisiones dependiendo del valor de alguna variables. Queremos dotar a nuestros programas de la capacidad de ejecutar instrucciones como esta: *si la variable `a` es par hacemos una cosa y si `a` es impar hacemos otra distinta*. En esta sección vamos a aprender a traducir ese tipo de instrucciones al lenguaje Python.

### 10.1. La instrucción if.

Ya hemos aprendido a seleccionar elementos de una lista según su posición. Pero a menudo la situación que se plantea es otra. Por ejemplo, volvamos al ejemplo del lanzamiento del dado 100 veces. Ahora que hemos aprendido cómo funciona `seed` la usaremos para que puedas reproducir exactamente nuestros ejemplos. Lanzamos el dado 100 veces (¿o son 100 dados que lanzamos a la vez? Conviene que te vayas haciendo estas preguntas):

```
In [1]: import random as rnd
In [2]: rnd.seed(2016)
In [3]: dado100 = [rnd.randrange(1, 7) for _ in range(0, 100)]
In [4]: print(dado100)
[6, 4, 5, 3, 6, 6, 1, 3, 2, 1, 4, 4, 1, 3, 2, 2, 3, 2, 3, 2, 2, 2, 4, 4, 6, 3, 1,
6, 6, 5, 2, 3, 5, 6, 1, 3, 5, 2, 4, 3, 1, 6, 4, 1, 3, 4, 1, 2, 5, 5, 1, 4, 5, 2,
3, 6, 2, 3, 1, 4, 5, 1, 3, 4, 6, 6, 3, 6, 3, 2, 6, 3, 6, 1, 5, 6, 5, 1, 5, 6, 6,
6, 4, 5, 2, 3, 2, 5, 4, 5, 5, 1, 1, 5, 2, 3, 4, 5, 5]
```

Fíjate, antes de seguir, en que son los mismos 100 valores que antes, porque hemos usado la misma semilla en `seed`.

Y ahora volvamos al trabajo. Si te pido que selecciones los primeros 10 valores de esta lista, sabes cómo hacerlo. ¿Pero y si te pido que selecciones los valores estrictamente mayores que 2? ¿O los valores pares? En ambos casos la selección no se basa en la posición sino en el valor (en particular, no confundas “ser un valor par” con “ser un valor que ocupa una posición par en la lista”).

Veamos el primer ejemplo: seleccionar los valores mayores que 2. La forma más sencilla de hacer esto en Python es con una comprensión de listas, añadiendo esa condición al final:

```
In [5]: mayorQue2 = [valor for valor in dado100 if valor > 2]
In [6]: print(mayorQue2)
[6, 4, 5, 3, 6, 6, 3, 4, 4, 3, 3, 3, 4, 4, 4, 6, 3, 6, 6, 5, 3, 5, 6, 3, 5, 4, 3, 6,
4, 3, 4, 5, 5, 4, 5, 3, 6, 3, 4, 5, 3, 4, 6, 6, 3, 6, 3, 6, 5, 6, 5, 5, 6,
```

La parte nueva del código es `if valor > 2`. Python utiliza `if` para introducir condiciones. En este caso la condición `valor > 2`. Y como ves, el resultado es una lista que contiene precisamente eso: los valores de `dado100` que son mayores que 2 (fíjate en que se respeta el orden de aparición de los valores).

Como siempre, también es posible hacer esto usando un bucle `for`. Y de hecho es interesante hacerlo para ver las diferencias entre ambos enfoques. Hemos pegado un bloque de código completo en la *Terminal* para obtener esto:

```
In [7]: mayorQue2 = []
....: for valor in dado100:
....:     if valor > 2:
....:         mayorQue2.append(valor)
....:

In [8]: print(mayorQue2)
[6, 4, 5, 3, 6, 6, 3, 4, 4, 3, 3, 3, 4, 4, 6, 3, 6, 6, 5, 3, 5, 6, 3, 5, 4, 3, 6,
4, 3, 4, 5, 5, 4, 5, 3, 6, 3, 4, 5, 3, 4, 6, 6, 3, 6, 3, 6, 5, 6, 5, 5, 6,
6, 6, 4, 5, 3, 5, 4, 5, 5, 3, 4, 5, 5]
```

Para empezar observemos que el resultado es, desde luego, el mismo. Y ahora fíjate en las líneas

```
if valor > 2:
    mayorQue2.append(valor)
```

Esta estructura es el primer ejemplo que encontramos de una instrucción `if`. Este es de hecho el ejemplo más sencillo:

1. Hay una primera línea de cabecera en la que aparece la condición.
2. Debajo de esta línea aparece un bloque de código indentado; es decir, desplazado hacia la derecha con respecto a la línea de cabecera, al igual que sucedía con el bucle `for`.

Y el funcionamiento de esta instrucción `if` es sencillo: si la condición se cumple, se ejecuta el bloque indentado de código. Si no se cumple, ese bloque se ignora y el programa continúa en la primera línea después del código.

### Ejercicio 29.

1. Sin ejecutarlos, adivina lo que va a imprimir este bloque de código:

```
a = 5
if a < 3:
    print("Esta línea forma parte del bloque if")
print("Esta es la primera línea tras el bloque if")
```

2. Haz lo mismo con esta segunda versión:

```
a = 2
if a < 3:
    print("Esta línea forma parte del bloque if")
print("Esta es la primera línea tras el bloque if")
```

3. Ejecuta ambos bloques en la Terminal y comprueba los resultados.



## 10.2. Condiciones.

En el ejercicio 29 hemos usado la condición `a < 3` para controlar el comportamiento de la instrucción `if`. Ahora queremos fijarnos en esa condición en sí misma. Después de asignar el valor `a = 2` escribimos la condición en la *Terminal* y la ejecutamos:

```
In [1]: a = 2
```

```
In [2]: a < 3
```

```
Out[2]: True
```

La respuesta de Python es que la condición es cierta, **True** en inglés. Por contra:

```
In [4]: a = 4
```

```
In [5]: a < 3
```

```
Out[5]: False
```

En este caso la condición es falsa y Python le asigna el valor **False**. Tenemos que aprender a pensar en la condición como si fuera una pregunta: “¿Es la variable **a** menor que 3?”. La respuesta sólo puede ser “sí” o “no”, o dicho de otra manera “cierto” o “falso”. En Python (y en muchos otros lenguajes de programación), las respuestas a ese tipo de preguntas se representan con un tipo especial de variables, las **variables booleanas** (por el matemático G. Boole<sup>4</sup>). Una variable de tipo booleano, por tanto, sólo puede tomar dos valores, que son **True** (cierto) o **False** (falso). Como habrás imaginado, las variables de tipo booleano son esenciales en programación, porque son la clave para que los programas puedan tomar decisiones mediante instrucciones **if** y otras instrucciones similares.

En los ejemplos previos hemos usado el operador de comparación **<** para construir la condición. Puedes construir condiciones similares con **>**, **<=** o **>=**. Además a menudo necesitaremos preguntarnos si dos valores son iguales. Para esto no podemos usar el símbolo **=**, porque como sabes Python lo usa para las asignaciones. La solución que nos ofrece Python es utilizar el símbolo **==**. Veamos como funciona:

```
In [6]: a = 3
```

```
In [7]: a == 3
```

```
Out[7]: True
```

```
In [8]: a = 5
```

```
In [9]: a == 3
```

```
Out[9]: False
```

Primero asignamos (un único igual **=**) a la variable **a** el valor 3. Después comparamos (con dos iguales **==**) la variable **a** con 3 y el resultado es **True**. Si ahora asignamos a la variable **a** el valor 5 y volvemos a hacer la pregunta **a == 3** la respuesta es **False**.

### Ejercicio 30.

1. *Usa el operador **==** y una comprensión de lista para averiguar cuántos elementos de la lista **dado100** del Ejercicio 28 (pág. 73) son iguales a 4. Recuerda que debes usar **rnd.seed(2016)** para generar la lista.*
2. *Para practicar, haz lo mismo con un bucle **for**.*
3. *Compara el resultado con la frecuencia absoluta de 4 que obtuvimos en el Ejercicio 28 (pág. 73).*
4. *¿Cómo haríamos para seleccionar los elementos pares de la lista **dado100**? Indicación: el operador **%** proporciona el resto de la división entera.*

□

También existe el operador *distinto de*, que en Python se representa con **!=**. Así, por ejemplo el resultado de:

<sup>4</sup>Más información en [http://es.wikipedia.org/wiki/George\\_Boole](http://es.wikipedia.org/wiki/George_Boole)

```
3 != 5
```

es `True`, mientras que el de:

```
2 != (1 + 1)
```

es `False`.

### 10.3. Operadores booleanos. Los valores booleanos como unos y ceros.

Ya hemos dicho que los valores booleanos `True` y `False` son la base sobre la que se construye toda la toma de decisiones en Python y, en general, en cualquier programa de ordenador. Las decisiones se toman comprobando si se cumple una cierta condición, que, a veces, puede ser tan sencilla como las que hemos visto. Pero en cuanto esa condición sea un poco más complicada, necesitaremos más herramientas. Por ejemplo, si queremos comprobar si el valor de la variable `a` es a la vez más grande que 3 y menor que 6, entonces necesitamos aprender una forma de expresar la conjunción “Y” en el lenguaje Python. En ese lenguaje, esto se expresa así:

```
(a > 3) and (a < 6)
```

El operador `and` es, en Python, el Y booleano. En otros lenguajes de programación se escribe de diferentes maneras (a menudo se usa `&`). Hemos ejecutado varios comandos en la *Terminal* para ilustrar el funcionamiento del operador `and`. Asignamos distintos valores a la variable `a` y para cada uno de ellos evaluamos la condición que hemos escrito más arriba:

```
In [1]: a = 4
```

```
In [2]: (a > 3) and (a < 6)  
Out[2]: True
```

```
In [3]: a = 1
```

```
In [4]: (a > 3) and (a < 6)  
Out[4]: False
```

```
In [5]: a = 6
```

```
In [6]: (a > 3) and (a < 6)  
Out[6]: False
```

Si examinas los resultados verás que de hecho esa condición caracteriza los valores de `a` que están entre 3 y 6 (estrictamente). El operador `and` es un **operador booleano**. Eso quiere decir que combina dos valores booleanos para dar como resultado otro booleano. De la misma forma que hemos aprendido tablas de multiplicar y sabemos que

$$2 \cdot 3$$

es 6 ahora podemos decir que

```
True and True
```

es (da como resultado) `True`. De hecho *la tabla completa* del operador `and` es esta (en la *Terminal*):

```
In [7]: True and True  
Out[7]: True
```

```
In [8]: True and False  
Out[8]: False
```

```
In [9]: False and True  
Out[9]: False
```

```
In [10]: False and False  
Out[10]: False
```

Si lo piensas un momento te darás cuenta de que esas cuatro son todas las situaciones posibles, por eso hemos dicho que es la tabla completa. Por ejemplo, al evaluar una condición como:

```
(3 < 5) and (6 < 4)
```

el primer paréntesis da como resultado `True` y el segundo `False`, así que (estamos en el segundo caso de la “*tabla*”) el operador `and` produce `False`. ¡Compruébalo!

Como ves, para que el resultado de `and` sea `True` es necesario que ambos operandos sean `True`. Otro operador booleano de Python, estrechamente emparentado con `and`, es el operador `or`, llamado O booleano. El operador `or` combina dos condiciones de manera que el resultado de `or` es `True` cuando *al menos una* de las condiciones es `True`. Se trata por tanto de un uso *no exclusivo* de la conjunción “o”. En cualquier caso, todo lo que se necesita saber de este operador se resume en estas cuatro situaciones posibles:

```
In [11]: True or True  
Out[11]: True
```

```
In [12]: True or False  
Out[12]: True
```

```
In [13]: False or True  
Out[13]: True
```

```
In [14]: False or False  
Out[14]: False
```

Y después de observarlas no deberías tener problemas en hacer el siguiente:

### Ejercicio 31.

¿Cuál es el resultado de la siguiente operación?

```
(7 < 5) or (2 < 4)
```



En resumen, el comportamiento de `or` es complementario al de `and`. Para que el resultado de `or` sea `False` es necesario que ambos operandos sean `False`.

### Evaluación de operaciones booleanas.

Opcional: esta sección puede omitirse en una primera lectura.

Aunque los valores booleanos básicos son `True` y `False`, Python utiliza una serie de reglas para convertir otro tipo de valores en booleanos. Por ejemplo, a primera vista puede parecer que la expresión:

```
3 and 5
```

no tiene sentido. El operador `and` es booleano, mientras que 3 y 5 son valores enteros. Sin embargo:

```
In [15]: 3 and 5  
Out[15]: 5
```

¿Qué significa esto? Python está aplicando dos reglas sencillas:

1. Todos los números enteros, salvo el cero, son equivalentes a `True`. El 0 es equivalente a `false`.
2. Cuando evalúa una condición lógica con enteros, Python aplica lo que se llama evaluación perezosa (en inglés *lazy evaluation*). Eso significa que sólo se evalúa la parte de la expresión booleana que es necesario evaluar para obtener la respuesta final. Y en ese momento se devuelve como resultado el último operando examinado.

Veamos como funciona esto en el ejemplo de `3 and 5`. Python empieza interpretando el `3` como `True`. En ese momento tenemos `True and ...` y si consultas la tabla de `and` te darás cuenta de que aún no podemos saber el resultado final. Así que Python necesita evaluar el siguiente operando, el `5`, que es equivalente a `True`. En este momento tenemos `True and True` y en principio el resultado sería `True`. Pero como el último valor examinado ha sido el `5`, ese es el que devuelve Python. Si todavía tienes dudas, puede que estos ejemplos te ayuden a entenderlo:

```
In [25]: 5 and 3  
Out[25]: 3
```

```
In [26]: 0 and 5  
Out[26]: 0
```

```
In [27]: 0 and 3  
Out[27]: 0
```

```
In [28]: 3 and 0  
Out[28]: 0
```

```
In [29]: 0 or 4  
Out[29]: 4
```

```
In [30]: 4 or 0  
Out[30]: 4
```

```
In [31]: 0 or 0  
Out[31]: 0
```

Los ejemplos que hemos visto utilizan valores enteros. Pero Python usa reglas similares de conversión a booleano para muchos otros tipos de variables. Por ejemplo:

```
In [32]: "azul" and "verde"  
Out[32]: 'verde'
```

Y, por otro lado:

```
In [33]: "azul" and ""  
Out[33]: ''
```

En efecto: Python interpreta cualquier cadena de texto como `True`, salvo la cadena vacía "", que se interpreta como `False`. Una vez entendido eso, el resto es igual que en el caso de los enteros. Creemos que es bueno que conozcas estas conversiones de valores a booleanos, porque algunos programadores las utilizan como *truco mágico* en algunos programas. Nosotros *desaconsejamos con énfasis* su uso: el resultado suele ser código difícil de interpretar y eso es siempre una mala idea (salvo que sea eso precisamente lo que se busca). Además otros lenguajes de programación pueden utilizar otras reglas. Hay, no obstante, otro tipo de conversiones mucho más interesantes y que si pueden ser de utilidad en Estadística. Y, de hecho, este segundo tipo de conversiones sí se hace igual en otros lenguajes de programación (como R). En algún sentido se trata del camino inverso. Si antes empezamos convirtiendo enteros a booleanos, debes saber que Python también puede interpretar los booleanos como enteros. Por ejemplo:

```
In [17]: True + False  
Out[17]: 1
```

Lo que ha hecho aquí Python es interpretar `True` como `1` y `False` como `0`. Aplicando ese principio es fácil entender lo que ha sucedido en cada uno de estos ejemplos:

```
In [18]: True * True  
Out[18]: 1
```

```
In [19]: True * False  
Out[19]: 0
```

```
In [20]: True - True
Out[20]: 0

In [21]: True / False
Traceback (most recent call last):

  File "<ipython-input-21-dcacc0b7b97e>", line 1, in <module>
    True / False

ZeroDivisionError: division by zero

In [22]: True / True
Out[22]: 1.0
```

Fíjate que Python lleva la interpretación hasta el final, de manera que dividir por `False` es dividir por 0.

Estos resultados permiten usar los valores booleanos para *contar* los elementos de una lista que verifican una condición. Por ejemplo, para saber cuántos elementos de la lista `dado100` que son menores que 5 y mayores que 1 podemos usar ese truco de los booleanos así:

```
In [1]: import random as rnd

In [2]: rnd.seed(2016)

In [3]: dado100 = [rnd.randrange(1, 7) for _ in range(0, 100)]

In [4]: sum([(valor > 1) and (valor < 5) for valor in dado100])
Out[4]: 48
```

Naturalmente, se puede hacer de esta otra manera, que tal vez ya hubieras pensado:

```
In [5]: len([valor for valor in dado100 if (valor > 1) and (valor < 5)])
Out[5]: 48
```

La decisión entre una u otra forma de trabajar es, a menudo, una cuestión de gustos.

## 11. Ejercicios adicionales y soluciones.

### Ejercicios adicionales.

28. En todos los ejemplos de aritmética vectorial que hemos visto, los dos vectores implicados eran de la misma longitud. ¿Qué sucede si no es así? Pues algo interesante, y que en cualquier caso conviene conocer para evitar errores. Te invitamos a que lo descubras ejecutando este ejemplo:

### Soluciones de algunos ejercicios.

- Ejercicio 2, pág. 6

```
In [31]: (1/3)+(1/5)
Out[31]: 0.5333333333333333

In [32]: 1 /((3+1)/5 )
Out[32]: 1.25
```

## Tutorial 03 (versión Python): Probabilidad.

Atención:

- Este documento pdf lleva adjuntos algunos de los ficheros de datos necesarios. Y está pensado para trabajar con él directamente en tu ordenador. Al usarlo en la pantalla, si es necesario, puedes aumentar alguna de las figuras para ver los detalles. Antes de imprimirla, piensa si es necesario. Los árboles y nosotros te lo agradeceremos.
- Fecha: 30 de mayo de 2016. Si este fichero tiene más de un año, puede resultar obsoleto. Busca si existe una versión más reciente.

## Índice

1. Los problemas del Caballero de Méré.	1
2. Ejemplos de Probabilidad Geométrica.	10
3. Operaciones simbólicas. Wiris, Wolfram Alpha.	15
4. Combinatoria en Python.	22
5. Combinatoria con Wiris y Wolfram Alpha	25
6. Ejercicios adicionales y soluciones	27

En la segunda parte del curso estamos aprendiendo el lenguaje matemático de la teoría de la Probabilidad que, ya en la tercera parte, nos va a resultar necesario para poder hacer Inferencia. En esta segunda parte del curso el nivel matemático se eleva, con la aparición de la Combinatoria, y de nociones del Cálculo, como las integrales. ¡Pero que nadie se asuste! En este tutorial vamos a tratar de poner los medios para aprender a calcular, de la forma más simple posible, lo que vamos a necesitar. Entre otras cosas, nos apoyaremos en el ordenador para evitarnos parte de los cálculos más difíciles.

### 1. Los problemas del Caballero de Méré.

Para poder hacer experimentos relacionados con la idea de probabilidad, necesitamos ser capaces de generar valores al azar. En los dos tutoriales previos, hemos avanzado algunas ideas de la forma en que podemos usar Calc y Python para esa tarea. Recordemos brevemente lo que hemos hecho hasta ahora:

- En la página 19 del Tutorial-01 hemos aprendido a usar la función `ALEATORIO.ENTRE` de Calc.
- En la Sección 9.1 del Tutorial-02 (pág. 65) hemos aprendido métodos básicos para generar números aleatorios en Python. Aquí vamos a aprovecharnos de esos métodos para hacer experimentos relacionados con la Probabilidad.
- También en ese Tutorial, concretamente en la Sección 9.5 (pág. 72), hemos aprendido a usar la función `seed` de Python para que los resultados sean reproducibles. En Calc puede hacerse

algo parecido con el *pegado especial*, pero es una herramienta mucho más incómoda que en Python<sup>1</sup>.

- Hemos insistido en la idea de que, en cualquier caso, los números que se generan con el ordenador son pseudoaleatorios (la propia existencia de `set.seed` es una prueba de esto). Pero podemos tranquilizar al lector: a todos los efectos, la diferencia entre estos números pseudoaleatorios y los números verdaderamente aleatorios es tan sutil, que no va a suponer ningún problema en este curso.

### 1.1. La función ALEATORIO.ENTRE de Calc y los problemas del Caballero de Méré.

La función ALEATORIO.ENTRE de Calc es suficiente para los primeros experimentos sencillos sobre la Regla de Laplace y, por ejemplo, los problemas del Caballero de Méré (ver la Sección 3.1, pág. 47 del libro). Para centrar la discusión, hemos aprendido que, por ejemplo,

`ALEATORIO.ENTRE(20;80)`

produce un número (pseudo)aleatorio entre 20 y 80. Con el lenguaje que estamos desarrollando en el Capítulo 3, podemos añadir que todos los resultados entre 20 y 80 son equiprobables. Si queremos conseguir más de un número, debemos copiar esa fórmula en más celdas de la hoja de cálculo. Cada vez que abras o cierres la hoja de cálculo, los números generados con ALEATORIO.ENTRE cambiarán. Y si deseas generar nuevos valores, sin tener que cerrar y abrir la hoja de cálculo, puedes utilizar la combinación de teclas **Ctrl+Mayús+F9**.

Vamos a usar la función para ilustrar los problemas del Caballero de Méré que hemos descrito en la Sección 3.1 (pág. 47). Concretamente, hemos adjuntado a este documento dos hojas de cálculo,

1. [Tut03-DeMere1.ods](#) (para la apuesta (a))
2. [Tut03-DeMere2.ods](#) (para la apuesta (b))

en las que hemos *simulado* esas dos apuestas, y hemos jugado 1000 veces cada una de ellas. La Figura 1 muestra el resultado al abrir el primero de esos ficheros. Tus números, al ser aleatorios, serán distintos de los nuestros, pero lo esencial de la discusión seguirá siendo válido. Las primeras cuatro

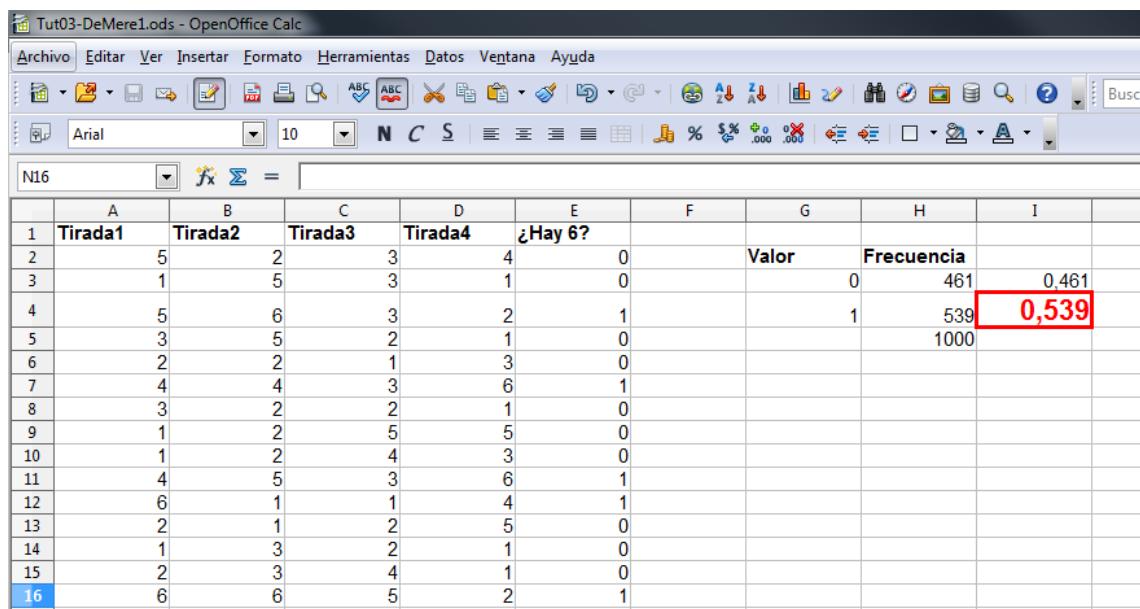


Figura 1: El fichero Tut03-DeMere1.ods de Calc

<sup>1</sup>Puedes ver los detalles en este enlace (en inglés):  
[https://wiki.openoffice.org/wiki/Documentation/How\\_Tos/Calc:\\_RAND\\_function](https://wiki.openoffice.org/wiki/Documentation/How_Tos/Calc:_RAND_function)

columnas de la hoja de cálculo (de la A a la D) se han obtenido usando la función **ALEATORIO.ENTRE**. Cada fila, por tanto se corresponde con una jugada, y si examinas el fichero verás que hemos jugado 1000 veces. Además, cada vez que pulses **Ctrl+Mayús+F9** obtendrás 1000 nuevas partidas de este juego. La columna E contiene un 1 si hemos obtenido (al menos) un 6 en las cuatro tiradas, y un 0 si no es así. No queremos entretenernos demasiado en la forma en la que hemos conseguido que Calc haga esto, porque haremos cosas más sofisticadas con Python, de una manera más sencilla, y merece la pena reservar nuestras fuerzas para ese empeño. Pero si sientes curiosidad, puedes hacer clic sobre la celda E2, y ver los comandos de Calc que hemos usado para conseguirlo:

```
=SI(0(A2=6;B2=6;C2=6;D2=6);1;0)
```

Esencialmente, lo que hemos dicho, en el lenguaje de Calc, es "Si alguna de las celdas A2, B2, C2, D2 contiene un 6, entonces el resultado es 1. En caso contrario, es un 0". Y para eso hemos usado dos funciones de Calc, llamadas **SI** y **0**.

Las columnas G, H e I contienen el análisis de los resultados en forma de tabla de frecuencias y frecuencias relativas. Las frecuencias relativas son las cantidades que debemos comparar con las probabilidades calculadas de forma teórica, para saber si la teoría de las probabilidades que estamos aplicando es una descripción correcta del fenómeno. Y, en este ejemplo en particular, el resultado nos indica que la teoría del Caballero de Méré no está funcionando.

Recuerda que, según hemos visto en la teoría del curso, la ganancia que el Caballero de Méré esperaba era de un 66 % de lo invertido. Y lo que se observa es que la proporción de apuestas perdidas frente a apuestas ganadas es mucho menor de lo que el Caballero esperaba.

En la Figura 2 puedes ver el comienzo del fichero correspondiente al segundo juego del Caballero de Méré. Sin entrar en muchos detalles (explora tú mismo el fichero!), las columnas de la izquierda contienen los resultados de las tiradas, que son números del 1 al 36, donde el 1 corresponde a (1, 1) y 36 corresponde a (6, 6). Y a la derecha aparece la tabla de frecuencias relativas, que muestra que el resultado es claramente distinto del que hemos obtenido en el otro juego. De hecho, la probabilidad de ganar en este segundo juego es aún más baja que en el otro (es aproximadamente igual a 0.49). Usa **Ctrl+Mayús+F9** unas cuantas veces para comprobarlo. Y recuerda que el Caballero de Méré creía que la probabilidad de ganar era la misma en ambos juegos. En la página 81 del Capítulo 3

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF
1	Tirada	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	¿Cuántos 6?	¿Hay 6?					
2		34	26	36	24	20	12	27	9	24	20	26	34	30	4	20	4	22	7	14	24	22	12	1	12	1	1	1	1			
3		1	21	21	27	6	21	9	23	9	16	7	18	21	29	1	24	30	31	2	3	12	4	15	15	0	0	0	0	0	0	
4		16	23	1	14	27	11	6	3	25	33	36	25	34	26	19	1	21	30	15	10	18	22	15	26	18	1	1	1	1	1	1
5		28	6	15	25	22	9	18	33	7	28	29	10	14	6	2	21	19	6	34	11	1	27	22	33	0	0	0	0	0	0	
6		36	29	12	13	4	8	12	16	12	22	24	24	4	2	38	18	15	3	32	33	14	11	26	18	13	2	1	1	1	1	1
7		15	12	4	21	30	21	24	23	10	35	26	23	18	26	7	9	14	32	30	34	3	23	14	18	0	0	0	0	0	0	
8		25	26	6	33	36	13	20	25	5	18	16	3	27	15	3	17	1	22	7	38	13	9	24	36	11	3	1	1	1	1	1
9		11	13	25	31	30	31	36	19	32	14	21	35	32	3	20	19	1	11	26	22	12	2	18	35	1	1	1	1	1	1	
10		26	35	10	8	17	20	32	7	4	6	1	12	3	18	15	6	24	1	7	32	10	18	31	1	0	0	0	0	0		
11		2	5	12	24	27	24	29	3	23	12	21	3	8	33	18	10	31	5	21	21	15	27	18	19	0	0	0	0	0	0	
12		26	3	10	21	21	28	20	16	1	29	24	21	22	9	14	4	3	23	32	26	24	19	8	31	0	0	0	0	0	0	
13		29	32	28	24	24	12	10	8	7	23	5	26	5	25	20	21	6	17	18	10	26	9	20	19	16	0	0	0	0	0	0
14		7	2	1	24	34	7	24	26	7	35	28	27	32	15	11	6	35	18	1	14	10	18	31	22	0	0	0	0	0	0	
15		3	35	6	18	25	18	21	6	24	35	11	2	27	33	15	14	38	16	28	4	9	26	3	22	1	1	1	1	1	1	

Figura 2: El fichero Tut03-DeMere2.ods de Calc

del libro se explica cómo calcular las probabilidades correctas para ambos juegos y, más adelante en este tutorial, daremos los detalles computacionales necesarios.

## 1.2. Probabilidades y la función sample de Python.

**Advertencia sobre la forma de mostrar el código en los tutoriales:** Para tratar de facilitar tu trabajo, verás que a partir de ahora el código Python aparece sombreado en color azul oscuro y que, justo debajo, se muestra la salida que produce la ejecución de ese código, sombreada en azul más claro. Por ejemplo:

```
a = 7  
print(a**2)
```

49

Recuerda, en lo que sigue, que cuando hablamos de *ejecutar* un fragmento de código en Python estamos indicando que puedes usar directamente la *Terminal* de Spyder o copiar ese fragmento en el *Editor de Código* y ejecutarlo desde allí con las teclas de función. Es decir que si copias y pegas el bloque de código de color azul oscuro en el *Editor de Código* de Spyder y pulsas F5 deberías obtener el resultado que mostramos en azul claro (salvo las diferencias que puedan causar los números pseudoaleatorios si no se usa `seed`).

Vamos a empezar recordando algo de lo que hemos aprendido sobre la forma de generar números pseudoaleatorios con las funciones del módulo `random`. Si queremos fabricar 100 números pseudoaleatorios entre 20 y 80, usaríamos (como hemos visto en la Sección 9.1 del Tutorial-02) un código similar a este:

```
import random as rnd  
  
rnd.seed(2016)  
  
a = 20  
  
b = 80  
  
n = 100  
  
datos = [rnd.randrange(a, b + 1) for _ in range(0, n)]  
  
print(datos)  
import random as rnd  
  
rnd.seed(2016)  
  
n = 1000  
  
partidas = []  
for i in range(0, n):  
    partida = []  
    for j in range(0, 4):  
        partida.append(rnd.randrange(1, 7))  
    partidas.append(partida)  
  
print(partidas[0:5])
```

```
[67, 75, 48, 77, 55, 76, 73, 36, 76, 63, 79, 65, 25, 72, 38, 28, 24, 51, 78,  
50, 26, 39, 28, 29, 40, 33, 42, 33, 35, 30, 51, 48, 79, 65, 40, 25, 66, 60, 55,  
35, 37, 58, 75, 72, 69, 62, 22, 39, 56, 76, 32, 47, 40, 21, 60, 44, 24, 39, 68,  
48, 26, 31, 53, 57, 22, 44, 57, 32, 75, 80, 42, 70, 62, 34, 42, 23, 44, 53, 23,  
36, 51, 65, 65, 43, 63, 69, 43, 31, 60, 39, 64, 23, 59, 66, 56, 22, 57, 78, 67,  
75]  
[[6, 4, 5, 3], [6, 6, 1, 3], [2, 1, 4, 4], [1, 3, 2, 2], [3, 2, 3, 2]]
```

Hemos usado la función `seed` para que cuando ejecutes el código obtengas los mismos resultados que nosotros. Es recomendable experimentar ejecutando el código sin usar esa función para comprobar qué es lo que cambia. En cualquier caso, al ejecutar este código los resultados quedan guardados en la lista `datos`.

Al construirlos así todos los números entre 20 y 80 son equiprobables. Y, puesto que estamos hablando de Probabilidad, es natural que nos preguntemos: ¿cómo podemos fabricar valores que no sean equiprobables? En el Tutorial-02 nos las ingeniamos para conseguirlo por el procedimiento

de aplicar la función `rnd.choice` a vectores en los que había elementos repetidos. Recuerda el ejemplo que veíamos entonces (aquí los resultados concretos son otros, pero no importa):

Este método, nos podría servir para conseguir lo que queremos. Pero no resulta una forma cómoda de trabajar, en cuanto las cosas sean un poco más complicadas.

## Ejercicio 1.

**Ejercicio 1.** Una caja contiene 100 fichas, idénticas salvo por el número que aparece en ellas. De ellas, 35 fichas están marcadas con el número 1, 15 con el número 2, 10 con el número 3, 10 con el número 4 y el resto con el número 5. Queremos extraer 20 fichas de la caja:

- (a) con reemplazamiento.

(b) sin reemplazamiento.

  1. Escribe el código Python que permite obtener esas 20 fichas, usando las funciones `sample` y `choice`. Los resultados, para los casos (a) y (b), se guardarán en dos listas, llamadas `sorteo1` y `sorteo2`, respectivamente. Utiliza `rnd.seed(2016)` como primera línea de tu código, para poder comparar tus resultados con los nuestros.
  2. Obtén las tablas de frecuencias absolutas y relativas de ambos vectores.
  3. ¿Qué tipo de error se produce en Python si tratas de extraer 1000 fichas sin reemplazamiento?

*Solución en la página 29.*

1

Más adelante se nos planteará una pregunta relacionada con esta, pero ligeramente distinta, en la que el punto de partida son las probabilidades. Imagínate, como ejemplo, que de nuevo tenemos una caja con fichas marcadas del 1 al 5. No sabemos cuántas fichas de cada tipo contiene la caja, pero sí sabemos que la probabilidad de sacar un 1 es del 17.3 %, la de sacar un 2 es el 15.2 %, la de sacar un 3 es un 27.7 % y la de sacar un 4 es un 18.3 % (y con eso es suficiente; no hace falta decir cuál es la probabilidad de sacar un 5, ¿verdad?). Si queremos simular la extracción de una muestra de 100 fichas de la caja (con reemplazamiento, así que no hay que preocuparse de que nos quedemos sin fichas), ¿cómo podríamos hacerlo en Python de manera que el código tenga en cuenta esas probabilidades? Para poder responder a esto necesitamos todavía aprender algo más de lenguaje Python. En particular nos resultará muy útil aprender a construir nuestras propias funciones en Python. Cuando lo hayamos hecho volveremos sobre esta cuestión.

### 1.3. El primer juego del caballero De Méré en Python.

Para cerrar este apartado y preparar la discusión del siguiente, vamos a pedirle al lector que vuelva un momento a pensar en la Figura 1 (pág. 2). La tabla del fichero Tut03-DeMere1.ods de Calc representa el resultado de 1000 partidas del primer juego del Caballero de Méré. En ese juego, nosotros agrupamos cuatro tiradas del dado y las llamamos *una partida* del juego. Para representar esto en Python vamos a crear una lista para cada partida y guardaremos todas las partidas en una lista de listas. Y de hecho vamos a hacer esto de dos formas: primero usaremos un enfoque clásico de la programación con dos bucles for anidados (recuerda la Sección 9.3 del Tutorial02). Este método tiene la ventaja de que resulta sencillo apreciar su estructura. Después usaremos un método que muchos programadores calificarían de *más Pythonico*: una comprensión de listas anidada (que ya vimos en esa misma Sección 9.3 del Tutorial02).

Vamos con el primer método. El código, que puedes copiar y ejecutar es:

```

# Importamos el módulo random y usamos seed
import random as rnd

rnd.seed(2016)

# Fijamos el número n de partidas.
n = 1000

# Generamos la lista de partidas.
# Cada partida es una lista de cuatro numeros de 1 a 6.
partidas = []
for i in range(0, n):
    partida = []
    for j in range(0, 4):
        partida.append(rnd.randrange(1, 7))
    partidas.append(partida)

# Mostramos las primeras 5 partidas.
print(partidas[0:5])

```

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

Recuerda que hemos usado `rnd.seed`, así que tus resultados deben coincidir con estos. Fíjate también en que los *contadores* `i`, `j` de los dos bucles `for` no intervienen en ninguna operación y sólo sirven para llevar la cuenta del número de partida (en el caso de `i`) y de la tirada dentro de esa partida (en el caso de `j`). Otro detalle destacable es que inicializamos la lista `partidas` como una lista vacía antes de comenzar los bucles, e iniciamos una nueva lista `partida` en cada iteración del bucle `for` externo: la lista `partida` se *vacía* así en cada iteración para reutilizarla, guardando en ella los cuatro resultados de una nueva partida.

Como decíamos, la estructura del método es explícita en esta versión con bucles `for` anidados. Frente a estos, el siguiente código con comprensión de listas anidadas resulta indiscutiblemente más conciso:

```

# Generamos las mismas partidas con comprensión de listas.
# Empezamos reiniciando el generador pseudoaleatorio con seed.
rnd.seed(2016)

# Comprensión de listas:
partidas = [[rnd.randrange(1, 7) for _ in range(0, 4)] for _ in range(0, n) ]

# Y mostramos las 5 primeras para comprobar que coinciden:
print(partidas[0:5])

```

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

La razón por la que los resultados son los mismos es que al ejecutar otra vez `rnd.seed(2016)` hemos reiniciado el generador de números aleatorios de Python y *lo hemos colocado en el mismo punto*. Así que a medida que le pedimos números aleatorios a Python, obtenemos las mismas respuestas que antes. Fíjate en que las seis líneas de los bucles `for` anidados se han reducido a una única línea de código. Fíjate también en que al usar el símbolo `_` hemos indicado que los contadores no juegan ningún papel adicional.

**Ejercicio 2.** En la Sección 9.3 del Tutorial02 vimos un primer ejemplo de comprensión de listas anidadas. Pero allí sólo había una pareja de corchetes `[]`, mientras que aquí hemos usado dos parejas anidadas. ¿Por qué? ¿Qué sucede si eliminas la pareja interna de corchetes? Solución en la página 30. Recuerda que el ejercicio te resultará mucho más útil si te esfuerzas en resolverlo, sin mirar la solución demasiado pronto. □

Vamos a analizar los resultados de esas 1000 partidas. Para saber si ha ganado en una partida concreta, el Caballero de Méré tiene que preguntarse si alguno de los cuatro números de esa partida es un 6. La descripción en pseudocódigo de lo que tenemos que hacer es muy sencilla:

```
Crear una lista vacía llamada partidaGanadora.  
Para cada partida de la lista partidas:  
    Si la lista contiene al menos un 6:  
        Añadir True a partidaGanadora.  
    En caso contrario:  
        Añadir False a partidaGanadora.
```

Seguramente ya te imaginas como traducir casi todo esto a código Python. Sólo hay un detalle que nos falta, pero es el detalle esencial: ¿cómo vamos a decidir si la partida contiene al menos un 6? Hay varias maneras de hacerlo. Antes de seguir es bueno que lo intentes.

**Ejercicio 3.** *Busca una forma de usar Python para decidir si una lista contiene al menos un 6. Sugerencia: la idea más evidente es usar un bucle `for` para recorrer la lista y ver si cada uno de los elementos es o no un 6. Solución en la página 31.* □

**¡No sigas, si no has hecho este ejercicio!**

En el Tutorial02 hemos aprendido las ideas básicas sobre valores booleanos y su uso en sentencias del tipo `if`. Allí aprendimos a usar condiciones como `3 < 5` o como `a == 4`. Pero iremos viendo en este tutorial y a lo largo del curso que hay muchas otras operaciones de Python que dan como resultado un valor booleano `True` o `False`. Concretamente hay una, la operación `in`, que nos permite hacer exactamente la pregunta que necesitamos. Para ver cómo funciona hemos ejecutado varios ejemplos de uso de esta operación:

```
print(6 in [3, 6, 1, 2])
```

```
True
```

```
print(6 in [3, 3, 1, 2])
```

```
False
```

```
print(6 in [3, 6, 6, 2])
```

```
True
```

Usando ese operador `in` la traducción del pseudocódigo para fabricar la lista que identifica las partidas ganadoras es muy sencilla:

```
# Obtenemos una lista de booleanos que caracteriza a las partidas ganadoras
# que son las que contienen al menos un 6.
```

```
partidasGanadoras = []
for partida in partidas:
    if 6 in partida:
        partidasGanadoras.append(True)
    else:
        partidasGanadoras.append(False)
```

```
# Se muestran las 5 primeras para comprobar.
```

```
print(partidasGanadoras[0:5])
```

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

Compara estos resultados con el contenido de cada una de esas 10 primeras listas para comprobar que está funcionando como esperamos.

**Ejercicio 4.** Aunque el operador `in` nos ha permitido resolver esta cuestión de una forma sencilla, no es la respuesta a todas nuestras posibles preguntas. Por ejemplo, imagínate que queremos saber cuáles son las partidas que contienen exactamente dos seises. ¿Cómo lo harías? Solución en la página 31. □

Una vez obtenida la lista `partidaGanadora` el siguiente paso es preguntarnos cuántas partidas ganadoras hay en ella. Es decir, cuántas veces aparece el valor `True` en esa lista. Y como sucede a menudo, hay muchas formas de hacerlo. Una de las formas que se te puede ocurrir es hacer una tabla de frecuencia de los valores `True` y `False`:

```
# Tabla de frecuencia

import collections as cl
ganadorasCounter = cl.Counter(partidasGanadoras)
freqGanadoras = ganadorasCounter.most_common()
print(freqGanadoras)
```

```
[(True, 526), (False, 474)]
```

Otra forma de hacerlo consiste en aprovechar un truco que vimos en la Sección opcional 10.3 (pág. 78) del Tutorial02. Allí mostramos que si sumamos o multiplicamos valores booleanos, Python convierte `True` en 1 y `False` en 0. De esa forma, al sumar una lista de valores booleanos el número que obtenemos es el número de valores `True` en esa lista. Algunos ejemplos:

```
lista = [True, False, True, True, False, False, True]
print(sum(lista))

lista = [True, True, True, True, True, False, True]
print(sum(lista))

lista = [False, False, False, False, False, False, False]
print(sum(lista))
```

```
4
6
0
```

Así que para saber cuantas partidas ganadoras hemos obtenido basta con hacer:

```
# El mismo resultado sumando booleanos:
print(sum(partidasGanadoras))
```

```
526
```

Y el resultado coincide con lo que hemos obtenido usando tablas de frecuencia. A partir de ahí es muy fácil calcular la proporción de partidas ganadoras:

```
# Proporcion de partidas ganadoras
proporcionGanadoras = sum(partidasGanadoras) / n
print(proporcionGanadoras)
```

```
0.526
```

En el libro hemos adelantado que el verdadero valor de la probabilidad de ganar es aproximadamente 0.52, así que el resultado de esta simulación es una estimación razonablemente buena de esa probabilidad. Naturalmente, esto es sólo un ejemplo. Pero basta con volver a ejecutar el código (sin usar `rnd.seed`, desde luego) para obtener otro conjunto de partidas y seguir el experimento, constatando, como hicimos con Calc, que la probabilidad estimada por el Caballero de Mérē era errónea. Para facilitar esa tarea, hemos agrupado el código en el fichero adjunto [Tut03-deMere01.py](#).

### Ejercicio 5.

1. Abre el fichero en el Editor de Código de Spyder, busca las líneas en las que se ejecuta `rnd.seed` (hay dos) y comenta esas líneas. Recuerda que eso significa que debes añadir el símbolo `#` al principio de cada una de esas líneas. A continuación ejecuta varias veces el programa para ver los resultados que obtienes.
2. Cambia el valor de `n` que inicialmente es 1000 por un valor más grande. Por ejemplo, prueba `n = 10000`. Vuelve a ejecutarlo varias veces y observa los valores que obtienes ahora. ¿Ves alguna diferencia con los del caso anterior?

3. ¿A partir de qué valor de  $n$  la respuesta deja de parecerse instantánea?

Solución en la página 32. □

La contestación al último apartado de este ejercicio depende de muchos factores y, en particular, de la potencia de tu ordenador. El objetivo es que te des cuenta de que incluso las simulaciones sencillas como esta representan un cierto esfuerzo de cálculo. A medida que afrontes problemas más y más complicados ese esfuerzo irá siendo cada vez más pesado y probablemente te obligará a empezar a preocuparte por el espinoso asunto de la eficiencia del código que escribes. Hemos visto que a menudo hay más de una forma de resolver un problema en Python. Pero eso no significa que todas las soluciones sean igual de eficientes. Para no sobrecargarte de información vamos a posponer esta discusión para más adelante. Cuando sepamos un poco más de Python volveremos sobre este tema.

## 1.4. El segundo juego

En lugar de hacer una descripción igual de detallada del segundo juego, vamos a proponer al lector que lo aborde mediante un ejercicio.

### Ejercicio 6.

1. Construye una lista deMere02 con 1000 partidas, cada una formada por 24 tiradas de un "dado" de 36 caras. El número 36 corresponde al 6 doble.
2. Identifica las partidas ganadoras, aquellas que contienen al menos un seis doble.
3. Calcula la proporción de partidas ganadoras sobre el total de 1000 partidas.

Solución en la página 32. □

### 1.4.1. Comprobando experimentalmente la regla de Laplace

Uno de nuestros primeros ejemplos usando la Regla de Laplace, al final de la Sección 3.2 del libro, se refiere al juego en el que tiramos dos veces un dado y queremos calcular la probabilidad del suceso

$$A = \text{obtener al menos un seis en las dos tiradas.}$$

La probabilidad ingenua predice  $\frac{12}{36} \approx 0.33$  para la probabilidad  $P(A)$ . En cambio, la regla de Laplace predice  $\frac{11}{36}$ . Para comprobarlo experimentalmente hemos incluido aquí una hoja de cálculo de Calc, llamada

[Tut03-DeMere1a.ods](#)

en la que se ha simulado ese lanzamiento. Recarga los valores unas cuantas veces (con **Ctrl+Mayús+F9**), para ver lo que sucede.

**Ejercicio 7.** Escribe el código Python necesario para comprobar estos resultados. Debería resultar fácil, ahora que hemos visto como hacerlo para los dos juegos del Caballero de Méré. Solución en la página 32. □

## 2. Ejemplos de Probabilidad Geométrica.

En la Sección 3.3 (pág. 51) hemos incluido varios ejemplos de un tipo de problemas que se denominan de Probabilidad Geométrica. Para visualizar esos problemas, además de Python ahora vamos a utilizar los primeros ficheros GeoGebra del curso.

## 2.1. Probabilidad geométrica. Método de Montecarlo.

### 2.1.1. Un punto al azar en el segmento $[0, 1]$ .

Vamos a usar Python para tratar de ilustrar las ideas que han aparecido en el Ejemplo 3.3.2 del libro (pág. 52). En ese ejemplo hemos tomado un número  $n_0$  muy grande, por ejemplo  $n_0 = 100000$ , para definir  $n_0 + 1$  puntos homogéneamente repartidos en el intervalo  $[0, 1]$ . Usaremos Python para construir esos puntos. Para comprobar el resultado mostraremos los primeros y los últimos puntos.

```
# n0 es el numero de subintervalos (puntos menos uno)
n0 = 100000

puntos = [ numero / n0 for numero in range(0, n0 + 1)]

# imprimimos los primeros y los últimos puntos
print(puntos[0:5])
print(puntos[-5:])
```

```
[0.0, 1e-05, 2e-05, 3e-05, 4e-05]
[0.99996, 0.99997, 0.99998, 0.99999, 1.0]
```

Ahora vamos a elegir  $k$  de esos puntos. Se trata de muestreo con reemplazamiento:

```
# k es el numero de puntos elegidos
k = 10000

# importamos el módulo random y usamos seed para tener reproducibilidad
import random as rnd
rnd.seed(2016)

# y usamos choice con comprensión de listas para elegir k puntos
elegidos = [rnd.choice(puntos) for _ in range(0, k)]

# estos son los primeros 10 elegidos
print(elegidos[0:10])
```

```
[0.96721, 0.58812, 0.7242, 0.33621, 0.89504, 0.92243, 0.10743, 0.38888,
0.16677, 0.09388]
```

Vamos a ver que proporción de ellos pertenece al intervalo  $A = [2/3, 1]$ .

```
enA = [punto for punto in elegidos if ((punto >= 2/3) and (punto <= 1))]

# comprobemos que funciona
print(enA[0:10])
```

```
[0.96721, 0.7242, 0.89504, 0.92243, 0.9317, 0.95725, 0.82419, 0.73637, 0.79739,
0.86465]
```

Todos los puntos de la lista `enA` son mayores que  $2/3 \approx 0.66 \dots$

```
# Ahora calculamos la proporción de puntos elegidos que pertenecen al intervalo:
proporcion = len(enA) / k
print(proporcion)
```

0.3358

La proporción, como ves, es cercana al valor  $\frac{1}{3} = 0.3333333$  que pronosticábamos.

### 2.1.2. Lanzando dardos con GeoGebra. El método de Montecarlo para calcular áreas.

Para ilustrar el Ejemplo 3.3.3 del libro (pág. 54), vamos a utilizar el fichero GeoGebra:

[Cap03-MonteCarloAreaCirculo.ggb](#)

La parte (a) de la Figura 3 muestra lo que verás cuando lo abras con GeoGebra. El cuadrado inicial, al que llamaremos  $C_1$ , es de lado 4, así que su área es 16. El interior del cuadrado contiene cinco puntos, porque el valor del deslizador inicialmente es  $n = 5$ . Arrastra el deslizador con el ratón para ir viendo como se generan más puntos en el interior del cuadrado. Por ejemplo, la parte (b) de la Figura 3 muestra un ejemplo en el que se han generado  $n = 2800$  puntos. Si ahora marcas con el ratón la casilla rotulada “Área del cuadrado pequeño”, verás aparecer un cuadrado más pequeño, de color rojo y centrado en el cuadrado más grande, al que vamos a llamar  $C_2$ . El lado de este cuadrado más pequeño es 2, así que su área es 4. Pero vamos a fingir que no sabemos cuánto vale el área, y vamos a tratar de calcular ese valor lanzando dardos, y contando la proporción de dardos que hacen blanco en el cuadrado pequeño. La idea subyacente es que ese número de dardos que aciertan en  $C_2$  es proporcional al área de  $C_2$ , así que tenemos:

$$\frac{\text{Dardos en } C_2}{\text{Total de dardos (en } C_1\text{)}} = \frac{\text{Área de } C_2}{\text{Área de } C_1}$$

En la Figura 4 se muestra ese proceso. Puedes ver que hemos lanzado  $n = 3000$  dardos, y que la proporción de aciertos en  $C_2$  es 0.265. Así que usando la ecuación anterior, y sabiendo que el área de  $C_1$  es 16, estimamos que:

$$\text{Área de } C_1 \approx 16 \cdot 0.265 \approx 4.24$$

Que, sin ser impresionante, es una primera aproximación. Prueba a mover el deslizador para ver como, a medida que el número de puntos aumenta, las estimaciones son cada vez mejores. Puedes moverlos hasta  $n = 5000$ . Y una vez que llegues a ese valor, pulsa **Ctrl + R**. Cada vez que lo hagas, GeoGebra volverá a lanzar 5000 nuevos dardos, y podrás ver una nueva estimación del área de  $C_2$  (también puedes mover el deslizador ligeramente para conseguir lo mismo).

Todo eso puede estar muy bien, pero el lector estará pensando que ya sabemos calcular (y de forma exacta) el área de un cuadrado. Al fin y al cabo de ahí hemos sacado el área 16 de  $C_1$ , para empezar. Es cierto. Pero lo interesante empieza ahora, el cuadrado era sólo calentamiento. Desmarca la casilla del cuadrado, y marca la del círculo. Vamos a llamar  $C_3$  al círculo que aparece, y cuyo radio es 1. Imagínate de nuevo que no conocemos la fórmula para el área del círculo. El razonamiento es el mismo, y nos conduce a la relación:

$$\frac{\text{Dardos en } C_3}{\text{Total de dardos (en } C_1\text{)}} = \frac{\text{Área de } C_3}{\text{Área de } C_1 \text{ (es 16)}}$$

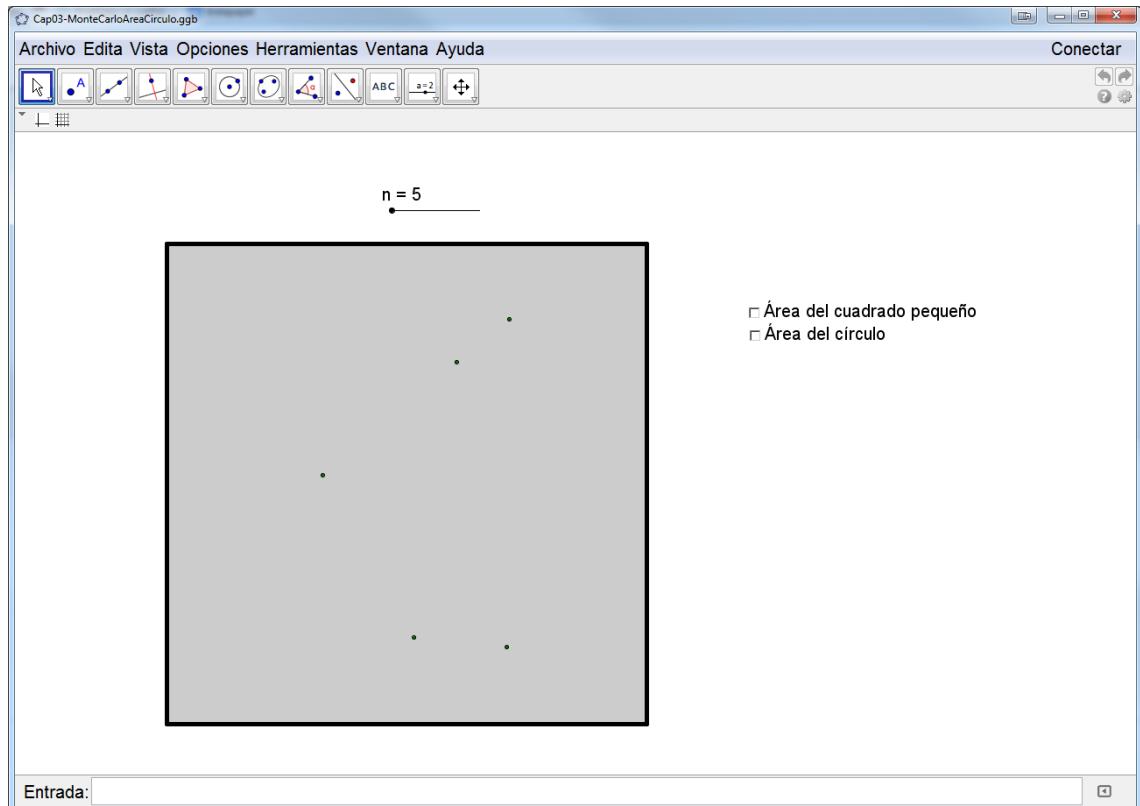
Por lo tanto,

$$\text{Área de } C_3 \approx 16 \cdot \frac{\text{Dardos en } C_3}{\text{Total de dardos (en } C_1\text{)}}.$$

Así que esto nos proporciona un procedimiento para aproximar el área del círculo (o de cualquier otra figura, por cierto) lanzando dardos. Eso empieza a parecer más interesante, ¿verdad?

Vuelve a repetir los pasos que dimos con el cuadrado, moviendo el deslizador hacia la derecha para ver como cambia la aproximación. Y cuando llegues a 5000 dardos, usa **Ctrl + R** para hacer varios

(a)



(b)

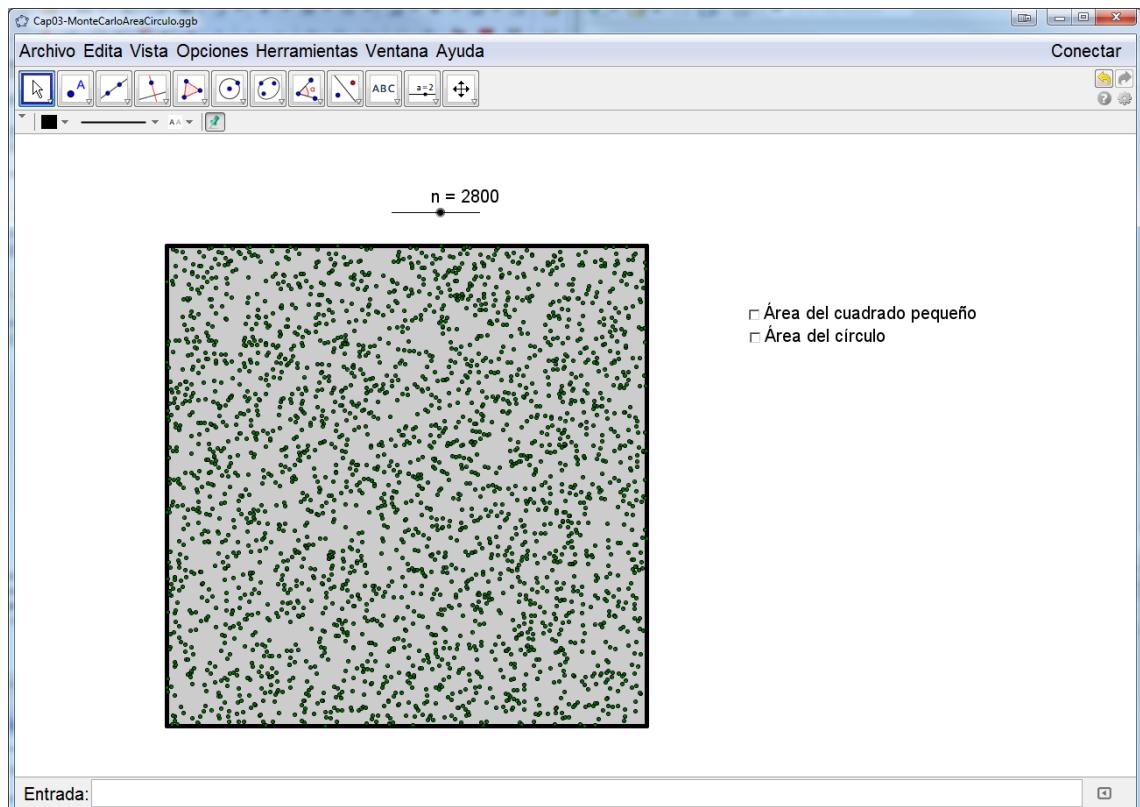


Figura 3: (a) Abriendo el fichero Cap03-MonteCarloAreaCirculo.ggb con GeoGebra (b) Más puntos al mover el deslizador...

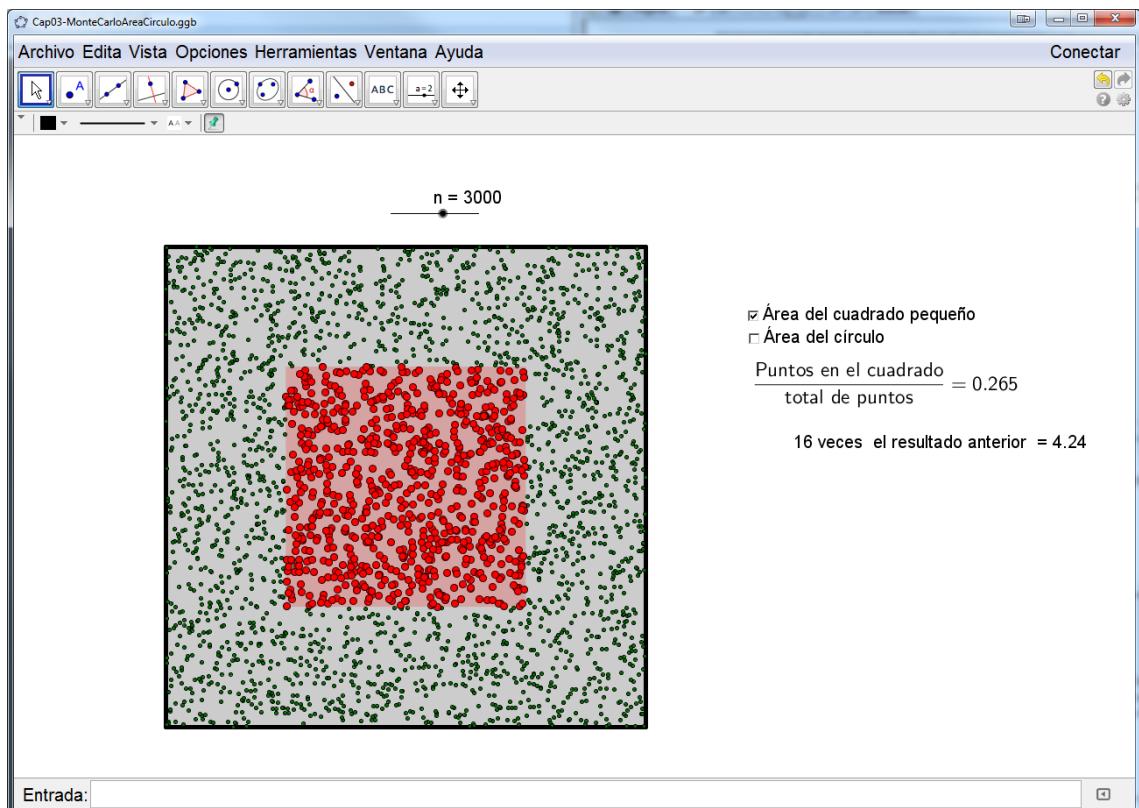


Figura 4: Lanzando dardos para calcular el área del cuadrado pequeño.

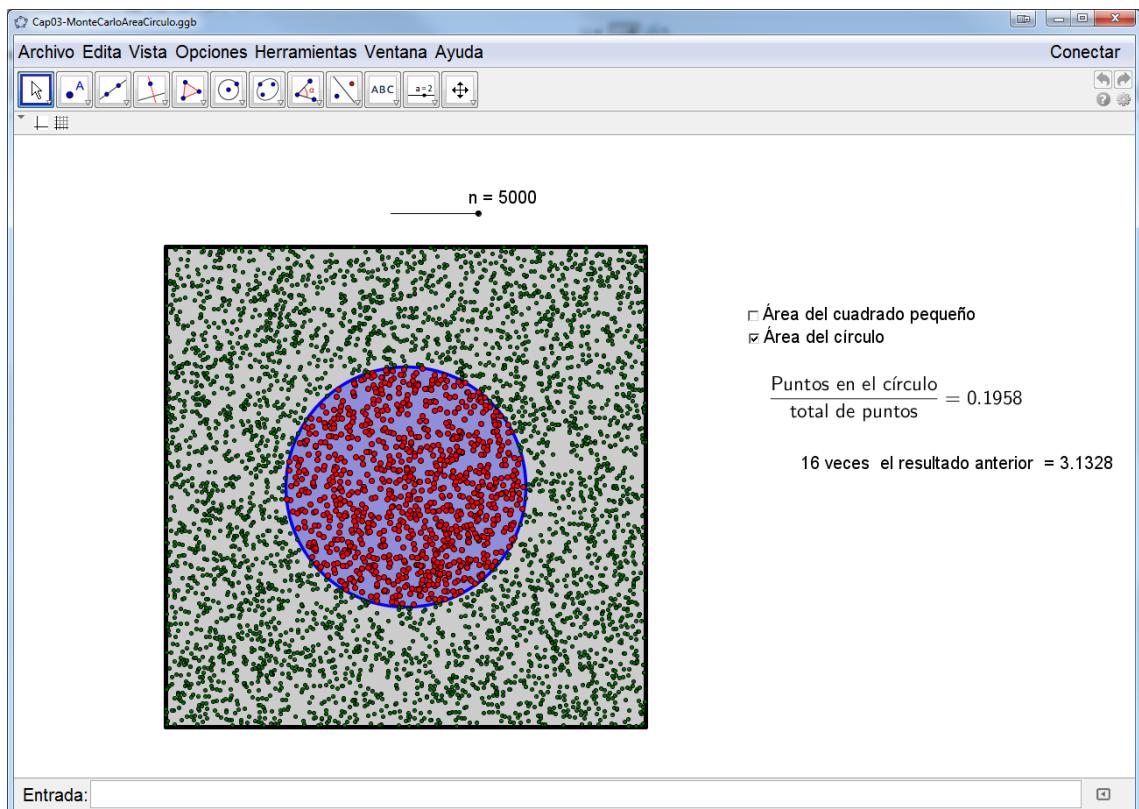


Figura 5: Lanzando dardos para calcular el área del círculo, que es precisamente  $\pi$ .

experimentos. Al cabo de unos cuantos intentos espero que te hayas convencido de que el área del círculo tiene un valor cercano a 3.1. El valor real, naturalmente, es  $\pi$ . Así que es posible calcular el

valor de  $\pi$  mientras juegas a los dardos (como ilustra la Figura 5)... siempre que estés dispuesto a jugar durante un buen rato, y no te esfuerces lo más mínimo en apuntar. Recuerda que lo esencial de estos experimentos es observar el vínculo que existe entre áreas y probabilidades. Cuanto mayor es el área de una figura, mayor es la probabilidad de acertarle con un dardo lanzado al azar.

### 3. Operaciones simbólicas. Wiris, Wolfram Alpha.

Cuando empezamos a trabajar con probabilidades, usando al principio la Regla de Laplace, y luego con la Combinatoria, a menudo nos surge la necesidad de operar con fracciones, como en el cálculo de esta fracción (que procede del Ejemplo 3.5.2, pág. 69 del libro):

$$\frac{\frac{3}{5} \cdot \frac{2}{6}}{\frac{3}{5} \cdot \frac{2}{6} + \frac{4}{5} \cdot \frac{4}{6}} = \frac{3}{11}$$

Si intentas hacer esta cuenta en Python, para empezar tienes que ser cuidadoso con los paréntesis (se aplica la regla de “más vale que sobren”), y escribir:

```
print( ( (3/5) * (2/6) ) / ( (3/5) * (2/6) + (4/5) * (4/6) ) )
```

```
0.2727272727272727
```

Fíjate en los espacios que hemos dejado para hacer esta expresión complicada más legible. La respuesta no es exactamente lo que queríamos. Para un problema como este, es muy posible que queramos ver el resultado en forma de fracción. El problema es que Python nos ha dado una respuesta numérica, usando la representación de los números en forma decimal. Podríamos usar algunos módulos de Python para trabajar con fracciones. Hay varias alternativas y la más sencilla seguramente sea utilizar el módulo `fractions`. Este módulo contiene una función llamada `Fraction` con la que podemos definir fracciones utilizando código como `Fraction(7, 12)` para representar la fracción  $\frac{7}{12}$ . Además una vez definidas esas fracciones podemos operar con ellas, en sumas, restas, productos y cocientes. Veamos cómo realizar la operación del ejemplo anterior con ese módulo `fractions`:

```
import fractions as fr

numerador = fr.Fraction(3, 5) * fr.Fraction(2, 6)
denominador = fr.Fraction(3, 5) * fr.Fraction(2, 6) + fr.Fraction(4, 5) * fr.Fraction(4, 6)
fraccion = numerador / denominador

print(fraccion)
```

```
3/11
```

Aunque no es necesario, hemos calculado por separado el numerador y denominador para hacer más legible el código. La respuesta es, como ves, la fracción

$$\frac{3}{11}.$$

Así pues, podemos trabajar con fracciones, al menos en algunas operaciones sencillas. Pero no resulta demasiado cómodo. A lo largo del curso nos vamos a encontrar varias veces con esta situación, en la que tenemos que trabajar con dos representaciones de un número: por un lado 0.2727273 es una versión numérica de la respuesta, en el sentido de *redondeada a unas cuantas cifras significativas* y, por lo tanto, es una respuesta aproximada. En cambio la respuesta en forma de fracción  $\frac{3}{11}$  es una respuesta simbólica, y es absolutamente exacta: **no hay ninguna pérdida de precisión**. Usamos la dualidad *numérico/simbólico* en el sentido habitual en las matemáticas contemporáneas. En ese sentido, las cantidades

$$\sqrt{2}, \quad \pi$$

son cantidades simbólicas, mientras que sus contrapartes numéricas (con cinco cifras significativas) son:

$$1.4142, \quad 3.1416$$

Y conviene insistir en que la representación numérica en un ordenador es una aproximación y conlleva una pérdida de precisión, que no sucede con las representaciones simbólicas. Por eso, para algunas operaciones del curso vamos a tener que recurrir a la ayuda de programas simbólicos. En las próximas secciones veremos algunos. ¿Por qué no trabajamos siempre con representaciones simbólicas, entonces? Pues porque los programas que las usan son mucho más complicados y en general mucho más lentos que los programas numéricos. A menudo se busca un equilibrio usando un enfoque híbrido, que combina parte simbólicas con otras numéricas dentro de un mismo cálculo.

### 3.1. Wiris.

El programa Wiris CAS (de Computer Algebra System) es una creación de la empresa de software matemático *Maths for More*, con sede en Barcelona, y fundada por profesores y antiguos estudiantes de la Universitat Politècnica de Catalunya. Para utilizar el programa debemos estar conectados a internet. El programa se usa en el navegador, a través de la Web, en la página de la propia empresa (que incluye publicidad insertada en la página):

<http://www.wiris.net/demo/wiris/es/index.html>

o a través de las páginas Wiris que algunas Consejerías de Educación de distintas comunidades autónomas españolas ponen a libre disposición del público (tras llegar a acuerdos con la empresa, claro). Aquí tienes, por ejemplo, el enlace de la Comunidad de Madrid:

<http://www.wiris.net/educa.madrid.org/wiris/es/index.html>

Usemos esta última. Al abrirla (es necesario tener instalado **Java** en el ordenador; si no sabes si lo tienes, o si puedes o debes instalarlo, consulta a tu ninja informático), el aspecto es este:



Wiris CAS nos permite escribir fórmulas matemáticas usando una notación muy parecida a la que usaríamos en el papel o la pizarra. Los símbolos y operaciones matemáticas están organizados por pestanas, pero para este primer ejemplo tan sencillo, todo lo que necesitamos está en la pestaña operaciones. Busca en ella el símbolo de fracción, que es el icono:



Púlsalo una vez con el ratón, y obtendrás

The screenshot shows the WIRIS calculator interface. At the top, there's a toolbar with various mathematical symbols like division, square root, summation, and product. Below the toolbar is a menu bar with options like Edición, Operaciones, Símbolos, Análisis, Matrices, Unidades, Combinatoria, Geometría, Griego, Programación, and Formato. The main workspace shows a fraction icon ( $\frac{\Box}{\Box}$ ) and an equals sign ( $=$ ). A cursor arrow is visible at the bottom right of the workspace.

A partir de aquí las cosas son bastante intuitivas. Tienes que usar las teclas + para la suma, y \* para la multiplicación, y usar el ícono de fracción cada vez que quieras crear una nueva fracción. Tus primeros pasos te pueden llevar a algo como esto:

The screenshot shows the WIRIS calculator interface again. The workspace now displays a more complex fraction:  $\frac{3}{5} \cdot \frac{\Box}{\Box}$ . The first fraction has a numerator of 3 and a denominator of 5. The second fraction has a numerator of  $\Box$  and a denominator of  $\Box$ . The equals sign ( $=$ ) is still present. The toolbar and menu bar are identical to the previous screenshot.

Ten en cuenta, para avanzar más rápido, que puedes seleccionar trozos de la fórmula con el ratón, y copiarlos y pegarlos. También, que en la pestaña Edición tienes dos iconos en forma de flechas curvas, para deshacer y rehacer operaciones. Debes llegar a:

The screenshot shows the WIRIS calculator interface. At the top, there's a menu bar with options like Edición, Operaciones, Símbolos, Análisis, Matrices, Unidades, Combinatoria, Geometría, Griego, Programación, and Formato. Below the menu is a toolbar with various mathematical symbols and functions. In the main workspace, there's a fraction multiplication problem displayed:

$$\frac{3}{5} \cdot \frac{2}{6} =$$

$$\frac{3}{5} \cdot \frac{2}{6} + \frac{4}{5} \cdot \frac{4}{6}$$

The result is shown as  $\frac{3}{11}$ .

Y ahora viene lo bueno. Una vez que estés ahí, pulsa sobre el icono *igual* que hay a la derecha de la fórmula (o, con el cursor situado en la fórmula, pulsa **Ctrl+Enter**). Al cabo de unos instantes (tu fórmula viaja por la red, se calcula, y la respuesta vuela de vuelta a tu ordenador), tendrás la respuesta:

The screenshot shows the WIRIS calculator interface again. The fraction multiplication problem is still present, but now the result is explicitly shown as  $\frac{3}{11}$ . A red arrow points from the original input to the result.

Y Wiris CAS está listo para nuestra siguiente pregunta. Como ves, la respuesta es simbólica, no numérica, como queríamos. El inconveniente que encontramos es que la respuesta es realmente una imagen en la pantalla, no un número que podamos cortar y pegar para llevar a otro programa.

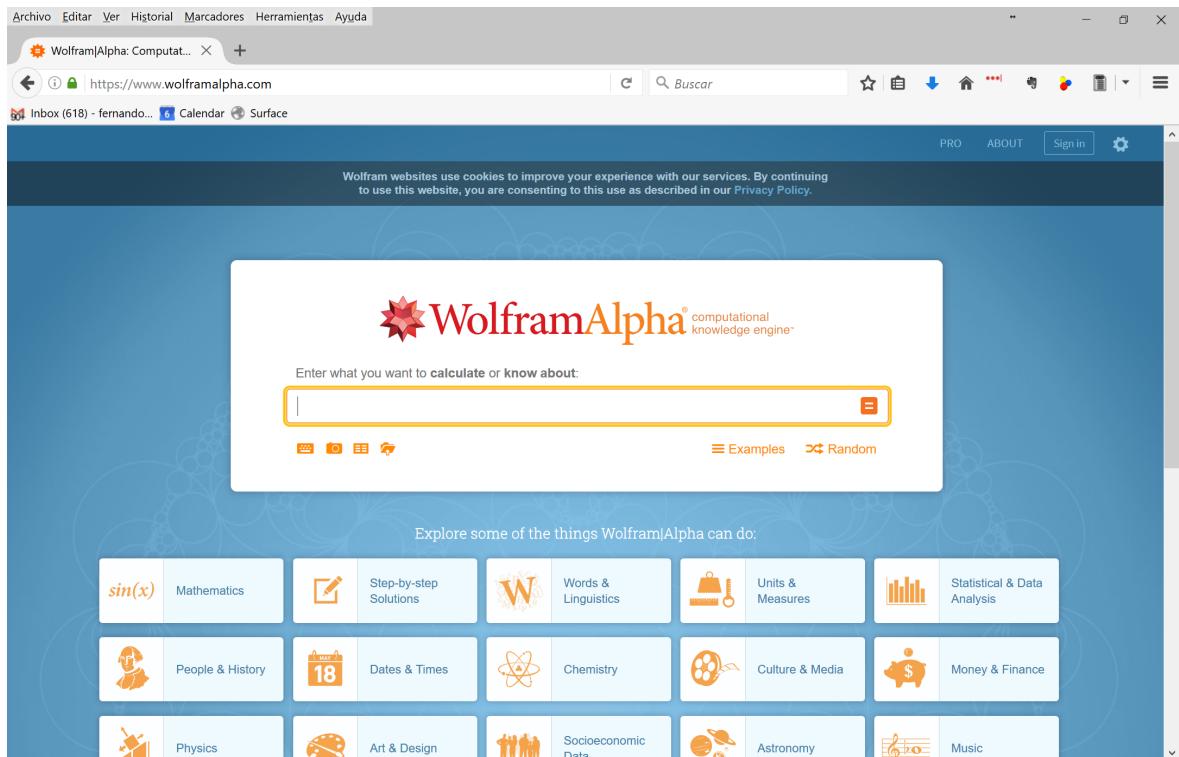
El cálculo de este ejemplo ha sido muy sencillo, pero a lo largo del curso iremos viendo más ejemplos en los que Wiris CAS nos puede ser de gran ayuda.

### 3.2. Wolfram Alpha.

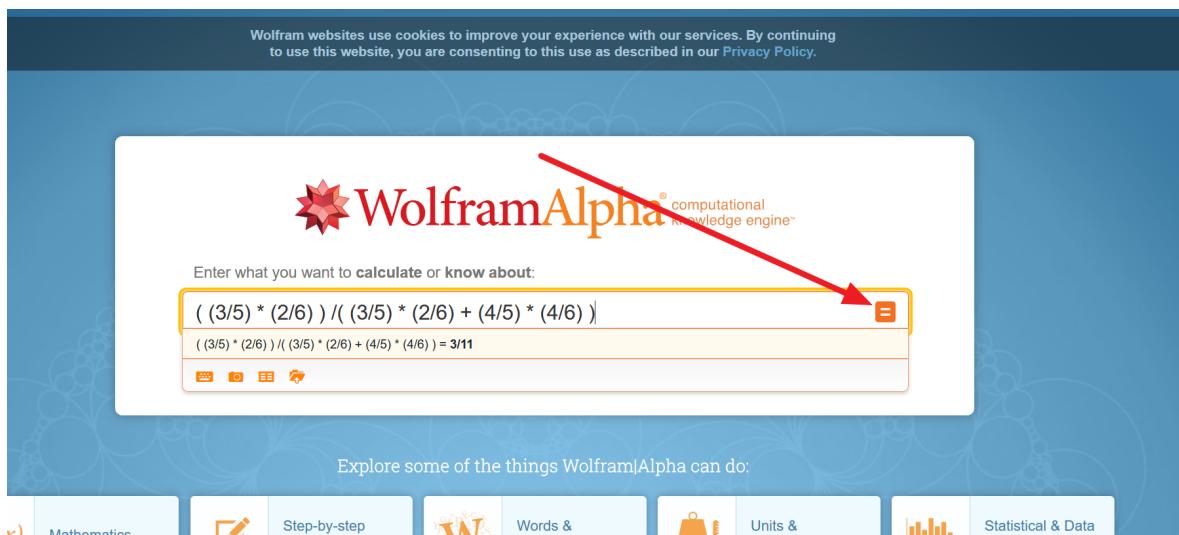
Se trata también de una herramienta sólo accesible a través de la Web, en este caso en inglés, pero que también es muy interesante (por ejemplo, es fácilmente accesible desde teléfonos móviles o tablets). Al abrir la dirección

<http://www.wolframalpha.com/>

Wolfram Alpha te recibe con una pantalla como esta:



Debemos introducir lo que queremos calcular en el campo de entrada del centro de la ventana, usando en este caso la misma sintaxis que en Python:



En este caso, como ves, en cuanto hemos tecleado la operación, Wolfram Alpha nos ha mostrado la respuesta. Pero para hacer el ejemplo más completo, pulsa sobre el símbolo igual que hay al final del campo de entrada. Verás una pantalla parecida a esta,

en la que, desde luego, está la respuesta a tu pregunta, pero que contiene además mucha más información matemática sobre esa pregunta (más de la que seguramente nunca pensaste que existiera...) Como en el caso de Wiris CAS, apenas hemos rozado la superficie de lo que Wolfram Alpha es capaz de hacer, y volveremos más adelante a seguir aprendiendo como usarlo. Si quieres, puedes pulsar en el enlace **Examples** que aparece bajo la barra de entrada para ver algunas de esas cosas.

### 3.3. Suma de series con Wiris y Wolfram Alpha.

Una serie es una suma con infinitos sumandos, como la suma

$$\frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^5} + \frac{1}{2^7} + \cdots = \frac{2}{3}.$$

que vimos en el Ejemplo 3.3.5 (pág. 58) del libro. La teoría matemática de este tipo de sumas puede llegar a ser muy complicada. Pero nosotros nos vamos a quedar en la superficie, limitándonos a usar el ordenador para calcular los ejemplos que necesitemos.

El primer ejemplo, el más sencillo de todos, nos permite comprobar que la probabilidad total asignada en ese Ejemplo 3.3.5 es igual a 1, como exigen las propiedades básicas de la Probabilidad. Es decir, que queremos ver que:

$$\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \cdots = 1.$$

En la primera suma de esta sección sumábamos sólo los términos con exponentes impares, aquí los sumamos todos. Para calcular esta suma usaremos Wiris. Una vez abierto, en la pestaña **Operaciones**, busca el ícono del sumatorio



Y usa la paleta de símbolos de esa misma pestaña para escribir la serie, como en la siguiente figura. Un par de advertencias:

- Encontrarás el símbolo de infinito positivo ( $+\infty$ ) en la pestaña **Símbolos**.
- Para escribir un exponente en Wiris puedes, desde luego, usar el botón  de la pestaña **Operaciones**. Pero es más rápido usar el atajo de teclado **Ctrl+↑**.

$$\left| \sum_{k=1}^{+\infty} \frac{1}{2^k} \right| =$$

Pulsa sobre el símbolo igual, y verás como Wiris te confirma que el resultado de sumar esa serie es un 1.

Ahora vamos a modificar ligeramente este cálculo, para obtener la suma de la serie de los términos con exponentes impares con la que empezamos esta sección. Para conseguir esto, vamos a modificar el exponente para que sólo tome valores impares. La forma de conseguirlo es sustituir, en el exponente, la variable  $k$  por la fórmula  $2 \cdot k - 1$ . Porque, cuando  $k$  recorre los valores  $1, 2, 3, \dots$ , la fórmula  $2 \cdot k - 1$  recorre a su vez los valores impares  $1, 3, 5, \dots$

Por lo tanto, hacemos ese cambio en Wiris, volvemos a pulsar el símbolo igual y obtenemos el resultado que se muestra en esta figura:

$$\left| \sum_{k=1}^{+\infty} \frac{1}{2^{2 \cdot k - 1}} \right| \rightarrow \frac{2}{3}$$

Como hemos dicho, las matemáticas de las series pueden ser muy complicadas. Queremos, para cerrar esta brevíssima visita, añadir un ejemplo final, que ilustra un punto importante de esa teoría. La suma de esta serie

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} + \dots$$

es infinito. Si le preguntas a Wiris, en este caso protestará, y dirá que no ha sido capaz de calcularlo (mira en la parte inferior de la pantalla). Lo malo es que a veces obtendremos esa misma respuesta con series que son “demasiado complicadas para Wiris”, pero cuya suma es una cantidad finita. Si alguna vez necesitas necesitas más detalles, puedes consultar el [manual de Wiris](#) al respecto.

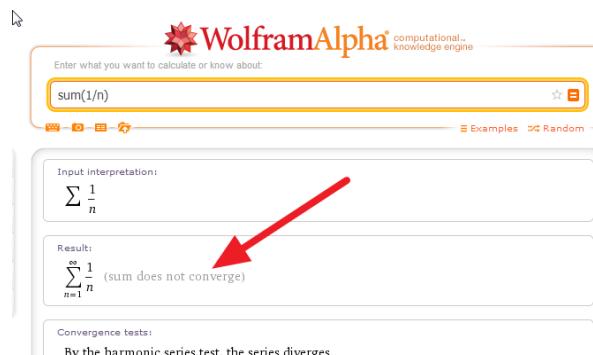
También puedes probar con Wolfram Alpha. Ya sabes, la dirección es

<http://www.wolframalpha.com/>

Prueba a escribir en el campo de entrada

sum (1/n)

y obtendrás esta respuesta



The screenshot shows the WolframAlpha search interface. In the search bar, the query "sum(1/n)" is entered. Below the search bar, the input interpretation is shown as  $\sum \frac{1}{n}$ . The result section contains the text "Result:  $\sum_{n=1}^{\infty} \frac{1}{n}$  (sum does not converge)". A red arrow points to this result text. At the bottom, under "Convergence tests:", it says "By the harmonic series test, the series diverges."

La frase `sum` does not converge (la suma –o serie– no converge) se debe, en este caso, a que el resultado no es un número finito<sup>2</sup>

¿Por qué esta suma es infinito, mientras que las anteriores daban resultados finitos? La respuesta es que una serie es una suma de infinitos números, cada vez más pequeños, y que el factor clave para que la suma sea finita es la *velocidad* a la que los números se hacen pequeños. Si se hacen pequeños muy rápido, la serie tendrá una suma finita (el resultado de la suma dependerá de cómo sea esa velocidad, en detalle). Pero si los números, por el contrario, aunque se hagan pequeños, lo hacen despacio, entonces la suma se irá haciendo cada vez más grande, hasta valer infinito, *en el límite*.

## 4. Combinatoria en Python.

### El factorial en Python.

Python nos proporciona numerosas herramientas para hacer cálculos en Combinatoria, desde las más básicas a algunas realmente sofisticadas. Para empezar, la función `factorial`, que se usa como muestra este ejemplo de código ejecutado en Jupyter:

```
import math as m  
  
print(m.factorial(6))
```

720

El factorial de un número entero  $n \geq 1$  da como resultado el producto de todos los números desde 1 hasta  $n$ . Otra manera de obtener ese mismo resultado, sin usar el módulo `math`, consiste en usar un bucle `for` para multiplicar esos números, como hemos hecho aquí:

```
fact = 1  
  
n = 6  
  
for i in range(1, n + 1):  
    fact = fact * i  
  
print(fact)
```

720

Nos hemos detenido en esto porque el factorial es sólo un caso especial del problema general de calcular el producto de una lista de números. Puesto que ya hemos visto varios casos en los que calculábamos la suma de todos los elementos de una lista de números (por ejemplo, al calcular la media aritmética) tal vez hayas pensado si no podríamos hacer algo parecido. Lamentablemente, Python no dispone de una función tan sencilla como la función `sum` que sirva para esto. Existen funciones que hacen eso, desde luego, pero siempre es necesario importarlas desde algún módulo.

**Ejercicio 8.** *El módulo numpy, que ya hemos usado antes, contiene una función prod. Busca información sobre esta función y úsala para calcular el factorial de 6. Solución en la página 32. □*

Puesto que casi todos los cálculos de Combinatoria que vamos a usar en el curso se pueden expresar usando el factorial, podríamos parar aquí. Pero el factorial es una forma extremadamente poco eficiente de hacer cálculos, que nos causará problemas en cuanto los números involucrados sean moderadamente grandes. Para trabajar de una forma más sensata tendremos que usar métodos que eviten los factoriales. A menudo usaremos funciones importadas de un módulo, que se han diseñado específicamente para hacer ese trabajo.

Vamos a aprovechar la ocasión para concoer a otro de los protagonistas habituales de la computación científica en Python, el módulo `scipy`. Puedes encontrar información sobre `Scipy` en:

<sup>2</sup>A parte de que el resultado sea infinito, pueden ocurrir otras cosas. La serie converge cuando el resultado es un número finito, y no converge en cualquier otro caso.

Este módulo se basa en `numpy` pero lo extiende añadiéndole una colección más amplia de objetos, funciones y algoritmos matemáticos. Por ejemplo, para calcular números combinatorios, el módulo `scipy` de Python nos ofrece la función `comb`. ¡Pero espera un poco antes de lanzarte! Este es el primer caso en el que nos vamos a encontrar con un módulo organizado en **submódulos**. La función `comb` pertenece al submódulo `misc` (de *miscellaneous*) dentro del módulo `scipy`. La forma de utilizar esta función para calcular el número combinatorio

$$\binom{22}{7}$$

es esta:

```
import scipy.misc as spm  
  
print(spm.comb(22, 7, exact=True))  
  
170544
```

#### 4.0.1. Construyendo una lista de permutaciones y combinaciones con Python.

Con las funciones que hemos visto hasta ahora podemos calcular el número de permutaciones de  $n$  objetos y también números combinatorios como  $\binom{10}{7}$ . Pero a veces podemos estar interesados en obtener explícitamente todas esas permutaciones o combinaciones. Insistimos en que no se trata de saber cuántos hay, sino de construirlos y enumerarlos. La forma más sencilla de hacer esto en Python es usar el módulo `itertools`, cuya documentación puedes encontrar en

<https://docs.python.org/3.5/library/itertools.html#module-itertools>

Entre otras funciones útiles este módulo incluye las funciones `permutations` y `combinations`. Por ejemplo, vamos a ver cómo utilizarlas para fabricar todas las listas posibles de tres elementos, elegidos de entre cuatro posibles. Para empezar, no admitimos repeticiones de los elementos. Entonces, si el orden no importa, estamos formando las combinaciones de cuatro elementos tomados de tres en tres. Y si el orden importa, entonces se trata de las variaciones (un inglés diría permutaciones) de cuatro elementos tomados de tres en tres (recuerda que en España, como hemos visto en el curso, distinguimos entre variaciones y permutaciones, pero la tradición anglosajona engloba dentro del término *permutations* tanto las variaciones como las permutaciones).

Los correspondientes comandos son estos. Para las combinaciones:

```
import itertools as it  
  
n = 4  
  
k = 3  
  
combinaciones = it.combinations(range(1, n + 1), k)  
  
print(combinaciones)
```

```
<itertools.combinations object at 0x10555b728>
```

Fíjate en que el resultado de usar `combinations` no es una lista, sino un objeto de tipo `itertools.combinations`. Lo convertimos en una lista usando `list` para poder manipular y mostrar el resultado cómodamente.

```
print(list(combinaciones))
```

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

Para las variaciones/permutaciones hacemos a continuación:

```
permutaciones = it.permutations(range(1, n + 1), k)

print(list(permutaciones))
```

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

Como cabía esperar, al considerar el orden hay muchas más permutaciones que combinaciones. Concretamente seis veces más (¿por qué?)

No hay nada en las funciones `permutations` y `combinations` que nos obligue a trabajar con listas de números. Podemos igualmente usar cadenas de caracteres, como en este ejemplo en el que construimos las seis permutaciones posibles de tres palabras:

```
import itertools as it

palabras = ["tres", "tristes", "tigres"]

permutaciones = list(it.permutations(palabras, 3))

print(permutaciones)
```

```
[('tres', 'tristes', 'tigres'), ('tres', 'tigres', 'tristes'), ('tristes',
'tres', 'tigres'), ('tristes', 'tigres', 'tres'), ('tigres', 'tres',
'tristes'), ('tigres', 'tristes', 'tres')]
```

En este caso hemos convertido el resultado de `permutations` directamente en una lista.

**Ejercicio 9.** ¿Qué ocurre con `combinations` y `permutations` si la lista original contiene elementos repetidos? Prueba con la lista:

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

y construye las combinaciones de los elementos de esta lista tomados de tres en tres. ¿Cuántos elementos tiene la lista que se obtiene como resultado? Y lo más importante ¿son todos distintos? Haz lo mismo para las permutaciones (advertencia: son muchas más, claro). Solución en la página 33. □

**Construyendo todas las posibles tiradas de un par de dados.**

Un problema relacionado con estos consiste en usar Python para construir la lista completa de tiradas de dos dados. Es decir, una lista que contenga los 36 pares posibles desde (1, 1), (1,2), hasta (6, 6). Fíjate que aquí se permiten repeticiones (así que no son variaciones) y además el orden es importante (no son combinaciones). De hecho se trata de construir lo que en el libro hemos llamado variaciones con repetición de 6 elementos tomados de dos en dos (ver pág. 80). El objeto resultante coincide con lo que en Matemáticas se denomina el producto cartesiano del conjunto {1, 2, 3, 4, 5, 6} consigo mismo. Por eso no es extraño que la función del módulo `itertools` que vamos a usar en este caso se llame `product`. La usamos así (fíjate en que hemos convertido el resultado de `product` directamente en una lista):

```

import itertools as it

tiradas = list(it.product(range(1, 7), repeat=2))

print(tiradas)

len(tiradas)

```

```

[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 1), (2, 2), (2, 3), (2,
4), (2, 5), (2, 6), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (4, 1), (4,
2), (4, 3), (4, 4), (4, 5), (4, 6), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5), (5,
6), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6)]

```

¿Qué papel juega el argumento `repeat=2` de la función? Descúbrelo haciendo este ejercicio:

#### Ejercicio 10.

1. Cambia `repeat=2` por `repeat=3` en ese código y ejecútalo. ¿Qué sucede?
2. Usa la lista `tiradas` que obtienes con `repeat=4` para calcular mediante la regla de Laplace la probabilidad de obtener al menos un seis en cuatro tiradas de un dado. Este método te permite obtener la probabilidad exacta (por ejemplo, usando Python para imprimir tanto el denominador como el numerador de la fracción que se obtiene con la Regla de Laplace). Y de esa forma, tendrás la solución correcta del primer problema del Caballero de Méré.

Solución en la página 33. □

## 5. Combinatoria con Wiris y Wolfram Alpha

En la Sección 3.6 (pág. 72) del libro hemos discutido la forma de calcular el número de permutaciones, variaciones, y combinaciones, con o sin repetición. Aquí vamos a aprender a utilizar algunas de las herramientas software que conocemos para facilitar el trabajo en problemas de Combinatoria.

### 5.1. Wiris

De nuevo en Wiris, en la pestaña de Combinatoria, encontrarás los iconos de esta figura:



Los significados son evidentes, así que nos limitamos a invitarte a que uses Wiris para calcular el resultado de estos ejercicios:

#### Ejercicio 11.

1. Permutaciones de 6 elementos.
2. Variaciones de 100 elementos, tomados de 30 en 30.
3. Combinaciones de 22 elementos, tomados de 7 en 7.
4. Permutaciones con repetición de 10 elementos, divididos en grupos de elementos idénticos de (2, 2, 3, 3) (de modo que hay 2 del primer tipo, 2 del segundo, 3 del tercero y 3 del cuarto tipo).
5. Variaciones con repetición de 8 elementos, tomados de 13 en 13.
6. Combinaciones con repetición de 4 elementos tomados de 6 en 6.

Soluciones en la pág. 34. □

A parte de calcular estos números, puedes usar Wiris para enumerar (es decir, hacer la lista, explícitamente) las variaciones o combinaciones. La forma de hacer esto es darle a Wiris como primer argumento una lista de los valores entre los que tiene que hacer la selección. Por ejemplo, para obtener la lista de variaciones de las 5 letras

$$\{a, b, c, d\}$$

tomadas de 2 en 2 (hay 12 posibles), hacemos esto (se muestra el resultado de ejecutar el comando)

The screenshot shows the Wiris interface with a menu bar at the top containing Edición, Operaciones, Símbolos, Análisis, Matrices, Unidades, Combinatoria, Geometría, Griego, Programación, and Formato. Below the menu, there are two input fields: one with 'C\_mn V\_mn P\_n' and another with 'CR\_mn VR\_mn P\_n''''. A command box contains '[ V\_{a,b,c,d}, 2 ]' followed by a red arrow pointing to the result '{ {a,b}, {b,a}, {a,c}, {c,a}, {a,d}, {d,a}, {b,c}, {c,b}, {b,d}, {d,b}, {c,d}, {d,c} }'.

Puedes aprender más sobre las capacidades combinatorias de Wiris en el manual:

<http://www.wiris.net/educa.madrid.org/wiris/manual/es/html/tour/combinatoria.html>

## 5.2. Wolfram Alpha

Para usar Wolfram Alpha, debemos expresar (en inglés, claro) lo que deseamos calcular en esa mezcla de lenguaje natural y símbolos matemáticos, característica de este sistema. Por ejemplo, si escribes en el campo de entrada:

number of permutations of 23 elements

obtendrás como respuesta

The screenshot shows the Wolfram Alpha interface. The search bar contains 'number of permutations of 23 elements'. Below the search bar are buttons for Examples and Random. The main area shows the input interpretation: 'permutations length 23'. Underneath, it displays the result: 'Number of distinct permutations of 23 objects: 25 852 016 738 884 976 640 000'. There are buttons for Approximate form, Download page, and a link to Wolfram Mathematica.

Si quieras copiar el resultado para poder pegarlo en otro programa, sitúa el ratón sobre ese resultado, y verás aparecer una barra de herramientas, como en la siguiente figura. Haz clic en la letra A para obtener una versión copiable como texto del resultado.

This screenshot shows the same Wolfram Alpha result as above, but with a cursor hovering over the numerical value '25 852 016 738 884 976 640 000'. A context menu has appeared with several options, including 'A' which is highlighted with a yellow background, indicating it's the option to copy the result as plain text.

Para ver más ejemplos de como usar Wolfram Alpha en Combinatoria, teclea **Combinatorics** en la línea de entrada. Ten en cuenta, en cualquier caso, que muchos de los ejemplos que verás no son relevantes para nuestro curso. Así que asegúrate de, al menos, hacer el siguiente

**Ejercicio 12.** Teclea en el campo de entrada de Wolfram Alpha los siguientes comandos y, para practicar, copia el resultado como texto en un editor de texto, como el Bloc de Notas.

1. permutations of 6 elements
2. permutations(100,30)
3. combinations(22,7)
4. number of permutations of a,a,b,b,c,c,c,d,d,d
5. Aunque no hay una sintaxis directa (al menos, yo no la conozco), usa Wolfram Alpha para calcular el número de variaciones con repetición de 8 elementos, tomados de 13 en 13, y el de combinaciones con repetición de 4 elementos tomados de 6 en 6.

Soluciones en la pág. 35. □

## 6. Ejercicios adicionales y soluciones

### Ejercicios adicionales

La mayor parte de los ejercicios adicionales de este tutorial no son *ejercicios de programación*, sino que están pensados para trabajar primordialmente pensando sobre el papel. Eso no significa que el ordenador no pueda servir de ayuda al pensamiento y, desde luego, como calculadora en muchos de esos ejercicios. Pero el énfasis está en la reflexión. Creemos que esta colección de ejercicios reúne un cierto equilibrio, sin adentrarse excesivamente en la Combinatoria más intrincada, pero la vez mostrando los casos particulares que más a menudo nos encontraremos en lo que resta del curso.

17. Se lanzan dos dados. Hallar la probabilidad de estos sucesos:
  - a) la suma de los resultados es ocho y (simultáneamente) su diferencia es cuatro.
  - b) la suma de los resultados es cinco y (simultáneamente) su producto es cuatro.
  - c) la suma de los resultados sea mayor que 12.
  - d) la suma de los resultados sea divisible entre 3.
18. Hallar la probabilidad de que al lanzar una moneda dos veces se obtenga al menos una vez una cruz.
19. En una caja hay seis fichas iguales numeradas del uno al 6. Se extraen una por una (sin reemplazarlas) todas las fichas de la caja. ¿Cuál es la probabilidad de que salgan en el orden natural? (Es decir, primero la ficha número uno, luego la dos, etc.)
20. En un paquete hay 20 tarjetas numeradas del 1 al 20. Se escogen al azar dos tarjetas. ¿Cuál es la probabilidad de que las dos que se han elegido sean la número 1 y la número 20? ¿Hay alguna diferencia entre sacar las dos tarjetas a la vez, o sacarlas consecutivamente sin reemplazamiento? ¿Y si es con reemplazamiento (sacamos una, la devolvemos al paquete y sacamos otra al azar)?
21. Una clase consta de 10 hombres y 20 mujeres. La mitad de los hombres y la mitad de las mujeres tienen los ojos castaños. Hallar la probabilidad de que una persona elegida al azar sea un hombre o tenga los ojos castaños.
22. Las siguientes afirmaciones son necesariamente falsas. Explica por qué.
  - a) En un hospital, la probabilidad de que un paciente permanezca ingresado durante más de dos días es de 0.5. La probabilidad de que un paciente permanezca hospitalizado durante más de un día es de 0.3.
  - b) La probabilidad de que llueva el sábado es del 50% y de que llueva el domingo es del 50%. Por tanto, durante el fin de semana es seguro que lloverá.
23. Se escogen al azar tres lámparas de entre 15, y sabemos que de esas 15, cinco son defectuosas. ¿Cuál es la probabilidad de que al menos una de las tres elegidas sea defectuosa?

24. Hallar la probabilidad de que al tirar tres dados aparezca el seis en uno de los dados (no importa cual), pero sólo en uno de ellos.
25. En una baraja de cartas española (40 cartas repartidas entre 4 palos) se desechan un número de cartas indeterminado. De las cartas que quedan se tiene una serie de probabilidades a la hora de sacar una carta:

$$P(\{\text{sacar rey}\}) = 0.15, \quad P(\{\text{sacar bastos}\}) = 0.3,$$

y además

$$P(\{\text{sacar carta que no sea rey ni bastos}\}) = 0.6.$$

- a) ¿Está entre las cartas no desecharadas el rey de bastos? En caso afirmativo, calcula la probabilidad de sacar esta carta.
- b) ¿Cuántas cartas hay?
26. En cierta facultad, se sabe que (1) un 25 % de los estudiantes suspendió Matemáticas, (2) un 15 % suspendió Química y (c) un 10 % suspendió ambas. Se selecciona un estudiante al azar:
- a) si suspendió Química, ¿cuál es la probabilidad de que también suspendiera Matemáticas?
- b) si suspendió Matemáticas, ¿cuál es la probabilidad de que también suspendiera Química?
- c) ¿Cuál es la probabilidad de que suspendiera al menos una de las dos?
27. En un experimento aleatorio, el suceso A tiene probabilidad 0.5, mientras que el suceso B tiene probabilidad 0.6. ¿Pueden ser los sucesos A y B incompatibles?
28. Un hospital tiene dos quirófanos en funcionamiento. En el primero se han producido incidentes en el 20 % de sus operaciones y el segundo sólo en el 4 %. El número de operaciones es el mismo en ambos quirófanos. La inspección hospitalaria analiza el expediente de una operación, elegido al azar y observa que en esa operación se produjo un incidente. ¿Cuál es la probabilidad de que la operación se realizara en el primer quirófano?
29. Un equipo de investigación está preparando un nuevo test para el diagnóstico de la enfermedad de Alzheimer. El test se ha probado en una muestra aleatoria con 450 pacientes diagnosticados con Alzheimer y una muestra aleatoria independiente de 500 pacientes que no presentan síntomas de la enfermedad. La siguiente tabla resume los resultados del ensayo:

		<u><b>Padecen Alzheimer</b></u>		
		Sí	No	Total
<u><b>Resultado del Test</b></u>	Positivo	436	5	441
	Negativo	14	495	509
	Total	450	500	950

Con estos datos, responder a las siguientes preguntas:

- a) ¿Cuál es la probabilidad de que un sujeto sano haya dado positivo en el test?
- b) ¿Cuál es la probabilidad de que un sujeto enfermo haya dado negativo en el test?
- c) Sabiendo que un sujeto ha dado positivo en el test, ¿cuál es la probabilidad de que esté enfermo?
- d) Sabiendo que un sujeto ha dado negativo en el test, ¿cuál es la probabilidad de que esté sano?
30. Una empresa produce anillas para identificación de tortugas marinas en tres fábricas. El volumen de producción diario es de 500, 1000 y 2000 unidades respectivamente. Se sabe que la fracción de producción defectuosa de las tres fábricas es de 0.005, 0.008, 0.010 respectivamente. Si se selecciona una anilla de forma aleatoria del total de producción de un día y se descubre que es defectuosa, ¿de qué fábrica es más probable que provenga esa anilla?.
31. Usa Python para simular el experimento que se describe en el Ejemplo 3.4.1 del libro (pág. 62). Por si te sirve de guía visual, en la hoja de cálculo (del programa Calc):

[Cap03-Lanzamientos2Dados-ProbabilidadCondicionada.ods](#)

se ha realizado una simulación para comprobar estos resultados.

Una extensión natural de este ejemplo es tratar de calcular  $P(S|D)$ . ¿Puedes modificar la hoja de cálculo para simular este otro caso?

32. Usa Python para simular el experimento que se describe en el Ejemplo 3.5.1 del libro (pág. 67), y que ilustramos con el fichero Calc

[Cap03-ProbabilidadesTotales-Urnas.ods](#)

33. Usa Python para simular los experimentos de los Ejemplos 3.6.3 y 3.6.4 del libro (pág. 78)
34. Elegimos al azar cinco números del 1 al 10, con reemplazamiento. Puedes pensar así: en una caja hay 10 bolas numeradas del 1 al 10. Sacamos una bola, anotamos el número, devolvemos la bola a la caja, y la agitamos bien. ¿Cuál es la probabilidad de que no haya repeticiones y, por tanto, obtengamos cinco números distintos?
35. De entre los números naturales 1, 2, ..., 20 se seleccionan cinco al azar sin reemplazamiento. Calcular la probabilidad de que: (a) los cinco sean pares. (b) exactamente dos de ellos sean múltiplos de 3. (c) dos sean impares y tres impares.
36. En la lotería primitiva gana quien acierta 6 números de entre 64 sin importar el orden en el que salgan. ¿Cuál es la probabilidad de ganar con una única apuesta?
37. De las 28 fichas del dominó, se extraen dos al azar (sin remplazamiento). ¿Cuál es la probabilidad de que con ellas se pueda formar una cadena, conforme a las reglas del juego (debe haber un número que aparezca en ambas fichas)?
38. Calcular la probabilidad de que un número de cuatro cifras (una matrícula, o un pin)
- tenga cuatro cifras diferentes.
  - tenga alguna cifra repetida.
  - tenga exactamente dos cifras iguales.
  - tenga dos parejas de cifras iguales (pero distintas entre sí).
  - tenga exactamente tres cifras iguales.
  - tenga todas las cifras iguales.
39. “**La paradoja del cumpleaños**”. Si en una sala hay 367 personas, entonces, con total seguridad, habrá dos personas con la misma fecha de cumpleaños (hemos usado 367 para cubrir incluso el caso de los años bisiestos, por si alguien de la sala nació el 29 de Febrero). Así que, si llamamos

$$A_n = \{\text{al menos dos de las } n \text{ personas cumplen años el mismo día}\}$$

entonces  $P(A_{367}) = 1$ . ¿Cuántas personas tiene que haber en la sala para que la probabilidad  $P(A_n)$  sea mayor que  $1/2$ ? Muchas menos de las que imaginas. Empieza por calcular ¿Cuánto vale  $P(A_5)$  usando el ejercicio anterior.

Este resultado se conoce como “la paradoja del cumpleaños”, aunque no tiene nada de paródico. Lo único que realmente demuestra este resultado es que, como hemos señalado en el curso, la intuición en materia de probabilidades es, en general, para la mayoría de nosotros, muy pobre.

## Soluciones de algunos ejercicios

- Ejercicio 1, pág. 5

```
import random as rnd
rnd.seed(2016)

valores = range(1, 6)
veces = [35, 15, 10, 10, 30]
```

```
caja = []
for i in range(0, len(valores)):
    caja = caja + [valores[i]] * veces[i]
print(caja)
```

```
conRemp = [rnd.choice(caja) for _ in range(20)]  
print(conRemp)
```

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

```
sinRemp = rnd.sample(caja, 20)  
print(sinRemp)
```

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

```
import collections as cl

freqConRemp = cl.Counter(conRemp).most_common()
print(sorted(freqConRemp))
```

`[(1, 9), (2, 4), (3, 1), (4, 2), (5, 4)]`

```
freqSinRemp = cl.Counter(sinRemp).most_common()  
print(sorted(freqSinRemp))
```

```
[(1, 7), (2, 2), (3, 2), (4, 1), (5, 8)]
```

- Ejercicio 2, pág. 6

Tenemos que incluir los corchetes para evitar que todos los elementos (4000 en total) terminen en una misma lista, sin separar las partidas. Eso es lo que ilustra el siguiente fragmento de código:

```
import random as rnd
rnd.seed(2016)

n = 1000

partidas2 = [rnd.randrange(1, 7) for _ in range(0, 4) for _ in range(0, n)]

print(partidas2[0:10])
```

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

- **Ejercicio 3, pág. 7**

Vamos a seguir trabajando con la misma lista `partidas`, generadas con una comprensión de listas:

```
import random as rnd
rnd.seed(2016)

n = 1000

partidas = [[rnd.randrange(1, 7) for _ in range(0, 4)] for _ in range(0, n) ]

print(partidas[0:5])

# Creamos una lista vacía en la que iremos
# guardando booleanos que para cada partida
# nos digan si contiene al menos un 6.
partidasGanadoras=[]

for partida in partidas:
    # Empezamos suponiendo que no hay un 6
    hayUn6 = False
    # Recorremos los resultados de la partida
    for resultado in partida:
        # Y si hay al menos un 6 cambiamos de opinión.
        if resultado == 6:
            hayUn6 = True
    # Al final añadimos la decisión a la lista.
    partidasGanadoras.append(hayUn6)

print(partidasGanadoras[0:5])
```

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

- **Ejercicio 4, pág. 8**

Una manera de hacerlo es esta:

```
# Vamos a buscar las partidas que contienen freqBuscada apariciones de cierto valor.
valor = 6
freqBuscada = 2
# Usamos la variable "coinciden" para llevar la cuenta del número de partidas
# en las que valor aparece freqBuscada veces.
coinciden = 0
# Ahora recorremos las partidas:
for partida in partidas:
    # Usamos comprensión de listas para ver la freq. del valor en esa partida.
    freqValor = len([item for item in partida if item == valor])
    # Y si aparece el número de veces deseado aumentamos el contador.
    if(freqValor == 2):
        coinciden = coinciden + 1
```

```
print(coinciden)
```

124

- **Ejercicio 5, pág. 9**

1. En las primeras 10 ejecuciones con las dos apariciones de `set.seed` comentadas yo he obtenido estos valores:

```
0.512, 0.51, 0.516, 0.537, 0526, 0.498, 0.532, 0.488, 0.523, 0.507
```

2. Y tras cambiar `n` por 10000 y ejecutar otras 10 veces he obtenido estos valores:

```
0.514, 0.5215, 0.5116, 0.5153, 0.5156, 0.5108, 0.5208, 0.519, 0.5054, 0.5217
```

Tus valores serán, desde luego, distintos de estos. Pero lo que deberías constatar, en cualquier caso, es que la *dispersión* de los valores ha disminuido mucho al aumentar `n`.

3. La respuesta a esta pregunta depende de muchas características de tu equipo, así que no podemos dar una respuesta general. Pero por si te sirve de referencia, en la máquina más potente a la que tenemos acceso el retraso entre el momento de la ejecución y el momento en el que aparece el resultado ocurre entre `n=30000` y `n=50000`.

- **Ejercicio 6, pág. 10**

Hemos incluido el código de una posible solución en el fichero adjunto [Tut03-deMere02.py](#).

- **Ejercicio 7, pág. 10**

El código de una posible solución, que es muy parecido al de los anteriores ejercicios, está en el fichero adjunto [Tut03-ejercicioReglaLaplace.py](#). El resultado de ejecutarlo es:

```
Estas son las cinco primeras partidas:  
[[6, 4], [5, 3], [6, 6], [1, 3], [2, 1]]  
Y esta indica cuáles de esas cinco primeras partidas son ganadoras:  
[True, False, True, False, False]  
El número de partidas ganadoras es:  
3071  
Y supone una frecuencia relativa de partidas ganadoras igual a:  
0.3071  
El valor predicho por la regla de Laplace es 11/36, que es aproximadamente =  
0.3055555555555555  
mientras la probabilidad ingenua predice 1/3, que es 0.333333.
```

Te aconsejo que tras ejecutarlo la primera vez y comprobar que obtienes lo mismo, comentes la línea de `set.seed` y ejecutes el código varias veces, con distintos valores de `n`. Yo he llegado hasta `n=500000` sin demasiadas dificultades.

- **Ejercicio 8, pág. 22**

```
import numpy as np  
print(np.prod(range(1, 7)))
```

720

- **Ejercicio 9, pág. 24**

Empezamos con las combinaciones:

```
import itertools as it  
  
datoCargado = [1, 2, 3, 4, 5, 6, 6, 6]  
  
combinaciones = list(it.combinations(datoCargado, 3))  
print("Lista de completa de combinaciones:")  
print(list(combinaciones))  
print("En total hay (¡no todas distintas!):")  
print(len(combinaciones))
```

```
Lista de completa de combinaciones:  
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 2, 6), (1, 2, 6), (1, 2, 6), (1, 3, 4),  
(1, 3, 5), (1, 3, 6), (1, 3, 6), (1, 3, 6), (1, 4, 5), (1, 4, 6), (1, 4, 6),  
(1, 4, 6), (1, 5, 6), (1, 5, 6), (1, 5, 6), (1, 6, 6), (1, 6, 6), (1, 6, 6),  
(2, 3, 4), (2, 3, 5), (2, 3, 6), (2, 3, 6), (2, 3, 6), (2, 4, 5), (2, 4, 6),  
(2, 4, 6), (2, 4, 6), (2, 5, 6), (2, 5, 6), (2, 5, 6), (2, 6, 6), (2, 6, 6),  
(2, 6, 6), (3, 4, 5), (3, 4, 6), (3, 4, 6), (3, 4, 6), (3, 5, 6), (3, 5, 6),  
(3, 5, 6), (3, 6, 6), (3, 6, 6), (3, 6, 6), (4, 5, 6), (4, 5, 6), (4, 5, 6),  
(4, 6, 6), (4, 6, 6), (4, 6, 6), (5, 6, 6), (5, 6, 6), (5, 6, 6), (6, 6, 6)]  
En total hay (¡no todas distintas!):  
56
```

Para las permutaciones obtenemos:

```
permutaciones = list(it.permutations(datoCargado, 3))  
print("Las primeras 10 permutaciones (fíjate en las repeticiones)")  
print(permutaciones[0:10])  
print("Las 10 últimas:")  
print(permutaciones[-10:])  
print("Y contando repeticiones hay:")  
print(len(permutaciones))
```

```
Las primeras 10 permutaciones (fíjate en las repeticiones)  
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 2, 6), (1, 2, 6), (1, 2, 6), (1, 3, 2),  
(1, 3, 4), (1, 3, 5), (1, 3, 6)]  
Las 10 últimas:  
[(6, 6, 3), (6, 6, 4), (6, 6, 5), (6, 6, 6), (6, 6, 1), (6, 6, 2), (6, 6, 3),  
(6, 6, 4), (6, 6, 5), (6, 6, 6)]  
Y contando repeticiones hay:  
336
```

Modifica el código para mostrarlas todas. Aquí sólo mostramos algunas por razones de espacio.

- **Ejercicio 10, pág. 25**

Para el primer apartado:

```

import itertools as it
tiradas = list(it.product(range(1, 7), repeat=3))
print(tiradas)
len(tiradas)

```

```

[(1, 1, 1), (1, 1, 2), (1, 1, 3), (1, 1, 4), (1, 1, 5), (1, 1, 6), (1, 2, 1),
(1, 2, 2), (1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 2, 6), (1, 3, 1), (1, 3, 2),
(1, 3, 3), (1, 3, 4), (1, 3, 5), (1, 3, 6), (1, 4, 1), (1, 4, 2), (1, 4, 3),
(1, 4, 4), (1, 4, 5), (1, 4, 6), (1, 5, 1), (1, 5, 2), (1, 5, 3), (1, 5, 4),
(1, 5, 5), (1, 5, 6), (1, 6, 1), (1, 6, 2), (1, 6, 3), (1, 6, 4), (1, 6, 5),
(1, 6, 6), (2, 1, 1), (2, 1, 2), (2, 1, 3), (2, 1, 4), (2, 1, 5), (2, 1, 6),
(2, 2, 1), (2, 2, 2), (2, 2, 3), (2, 2, 4), (2, 2, 5), (2, 2, 6), (2, 3, 1),
(2, 3, 2), (2, 3, 3), (2, 3, 4), (2, 3, 5), (2, 3, 6), (2, 4, 1), (2, 4, 2),
(2, 4, 3), (2, 4, 4), (2, 4, 5), (2, 4, 6), (2, 5, 1), (2, 5, 2), (2, 5, 3),
(2, 5, 4), (2, 5, 5), (2, 5, 6), (2, 6, 1), (2, 6, 2), (2, 6, 3), (2, 6, 4),
(2, 6, 5), (2, 6, 6), (3, 1, 1), (3, 1, 2), (3, 1, 3), (3, 1, 4), (3, 1, 5),
(3, 1, 6), (3, 2, 1), (3, 2, 2), (3, 2, 3), (3, 2, 4), (3, 2, 5), (3, 2, 6),
(3, 3, 1), (3, 3, 2), (3, 3, 3), (3, 3, 4), (3, 3, 5), (3, 3, 6), (3, 4, 1),
(3, 4, 2), (3, 4, 3), (3, 4, 4), (3, 4, 5), (3, 4, 6), (3, 5, 1), (3, 5, 2),
(3, 5, 3), (3, 5, 4), (3, 5, 5), (3, 5, 6), (3, 6, 1), (3, 6, 2), (3, 6, 3),
(3, 6, 4), (3, 6, 5), (3, 6, 6), (4, 1, 1), (4, 1, 2), (4, 1, 3), (4, 1, 4),
(4, 1, 5), (4, 1, 6), (4, 2, 1), (4, 2, 2), (4, 2, 3), (4, 2, 4), (4, 2, 5),
(4, 2, 6), (4, 3, 1), (4, 3, 2), (4, 3, 3), (4, 3, 4), (4, 3, 5), (4, 3, 6),
(4, 4, 1), (4, 4, 2), (4, 4, 3), (4, 4, 4), (4, 4, 5), (4, 4, 6), (4, 5, 1),
(4, 5, 2), (4, 5, 3), (4, 5, 4), (4, 5, 5), (4, 5, 6), (4, 6, 1), (4, 6, 2),
(4, 6, 3), (4, 6, 4), (4, 6, 5), (4, 6, 6), (5, 1, 1), (5, 1, 2), (5, 1, 3),
(5, 1, 4), (5, 1, 5), (5, 1, 6), (5, 2, 1), (5, 2, 2), (5, 2, 3), (5, 2, 4),
(5, 2, 5), (5, 2, 6), (5, 3, 1), (5, 3, 2), (5, 3, 3), (5, 3, 4), (5, 3, 5),
(5, 3, 6), (5, 4, 1), (5, 4, 2), (5, 4, 3), (5, 4, 4), (5, 4, 5), (5, 4, 6),
(5, 5, 1), (5, 5, 2), (5, 5, 3), (5, 5, 4), (5, 5, 5), (5, 5, 6), (5, 6, 1),
(5, 6, 2), (5, 6, 3), (5, 6, 4), (5, 6, 5), (5, 6, 6), (6, 1, 1), (6, 1, 2),
(6, 1, 3), (6, 1, 4), (6, 1, 5), (6, 1, 6), (6, 2, 1), (6, 2, 2), (6, 2, 3),
(6, 2, 4), (6, 2, 5), (6, 2, 6), (6, 3, 1), (6, 3, 2), (6, 3, 3), (6, 3, 4),
(6, 3, 5), (6, 3, 6), (6, 4, 1), (6, 4, 2), (6, 4, 3), (6, 4, 4), (6, 4, 5),
(6, 4, 6), (6, 5, 1), (6, 5, 2), (6, 5, 3), (6, 5, 4), (6, 5, 5), (6, 5, 6),
(6, 6, 1), (6, 6, 2), (6, 6, 3), (6, 6, 4), (6, 6, 5), (6, 6, 6)]

```

Para el segundo apartado, el código de una posible solución está en el fichero adjunto [Tut03-ejercicioDeMereExacto.py](#). El resultado de ejecutarlo es:

```

Primeras 10 tiradas:
[(1, 1, 1, 1), (1, 1, 1, 2), (1, 1, 1, 3), (1, 1, 1, 4), (1, 1, 1, 5), (1, 1, 1, 6),
(1, 1, 2, 1), (1, 1, 2, 2), (1, 1, 2, 3), (1, 1, 2, 4)]
y las 10 últimas:
[(6, 6, 5, 3), (6, 6, 5, 4), (6, 6, 5, 5), (6, 6, 5, 6), (6, 6, 6, 1), (6, 6, 6, 2),
(6, 6, 6, 3), (6, 6, 6, 4), (6, 6, 6, 5), (6, 6, 6, 6)]
El número de tiradas posibles es
1296
que coincide con el valor teórico 6**4
1296
El número de partidas ganadoras es:
671
Así que la probabilidad calculada por la regla de Laplace es la fracción:
671/1296
que es aproximadamente igual a
0.5177469135802469

```

- Ejercicio 11, pág. 25

1. 720.
2. 7791097137057804874587232499277321440358327700684800000000. Este número podría representar el número de subgrupos distintos de 30 alumnos, que podemos formar a partir de una clase de 100 alumnos. Su orden de magnitud es de  $10^{57}$ . Para que te hagas una idea, el número estimado de estrellas en el universo es del orden de  $10^{24}$ .
3. 170544.
4. 25200.
5. 549755813888.
6. 84.

• **Ejercicio 12, pág. 27**

Todas las preguntas aparecen en el Ejercicio 11; consulta las respuesta a ese ejercicio.

---

Fin del Tutorial-03. ¡Gracias por la atención!

## Tutorial 04 (versión Python): Variables aleatorias.

Atención:

- Este documento pdf lleva adjuntos algunos de los ficheros de datos necesarios. Y está pensado para trabajar con él directamente en tu ordenador. Al usarlo en la pantalla, si es necesario, puedes aumentar alguna de las figuras para ver los detalles. Antes de imprimirla, piensa si es necesario. Los árboles y nosotros te lo agradeceremos.
- Fecha: 1 de junio de 2016. Si este fichero tiene más de un año, puede resultar obsoleto. Busca si existe una versión más reciente.

## Índice

1. Variables aleatorias discretas con Python.	1
2. Funciones definidas por el usuario en Python.	10

### 1. Variables aleatorias discretas con Python.

#### 1.1. Tabla (función) de densidad de una variable aleatoria discreta.

Una variable aleatoria discreta  $X$  que toma los valores

$$x_1, x_2, \dots, x_k$$

se define mediante su tabla de densidad de probabilidad:

Valor:	$x_1$	$x_2$	$x_3$	...	$x_k$
Probabilidad:	$p_1$	$p_2$	$p_3$	...	$p_k$

Como ya hemos dicho, las probabilidades se pueden entender, en muchos casos, como una versión teórica de las frecuencias relativas. Así que esta tabla se parece mucho a una tabla de frecuencias relativas, y parte de las operaciones que vamos a hacer nos recordarán mucho a las que hicimos en la primera parte del curso usando tablas de frecuencias.

La primera variable aleatoria que vamos a estudiar va a ser, como en el Ejemplo 4.1.1 del libro (pág. 97), la variable  $X$  cuyo valor es el resultado de sumar los puntos obtenidos al lanzar dos dados. Para estudiarla, vamos a recordar algunas de las técnicas de simulación que aprendimos en el Tutorial03. Usaremos listas de Python para reproducir los resultados de ese Ejemplo 4.1.1. Concretamente, los resultados posibles al tirar el primer dado son:

```
dado1 = range(1,7)
print(list(dado1))
```

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

Ahora hacemos lo mismo para el segundo dado:

```
dado2 = range(1,7)
```

Las sumas posibles se obtienen entonces así usando una comprensión de listas:

```
suma = [d1 + d2 for d1 in dado1 for d2 in dado2]
print(suma)
```

```
[2, 3, 4, 5, 6, 7, 3, 4, 5, 6, 7, 8, 4, 5, 6, 7, 8, 9, 5, 6, 7, 8, 9, 10, 6, 7,
8, 9, 10, 11, 7, 8, 9, 10, 11, 12]
```

Para hacer el recuento de las veces que aparece cada uno de los valores posibles de la suma podemos usar los métodos que aprendimos en el Tutorial02 y hacer una tabla de frecuencias:

```
import collections as cl
sumaCounter = cl.Counter(suma)
recuentoValoresSuma = sumaCounter.most_common()
recuentoValoresSuma.sort()
print(recuentoValoresSuma)
```

```
[(2, 1), (3, 2), (4, 3), (5, 4), (6, 5), (7, 6), (8, 5), (9, 4), (10, 3), (11,
2), (12, 1)]
```

Para convertir estos recuentos en probabilidades tenemos que dividirlos por el número de resultados posibles, que son 36. Recuerda que el resultado de la división en Python nos va a proporcionar respuestas numéricas (no *simbólicas*) y por lo tanto, aproximadas:

```
n = len(suma)
print("n=", n)
probabilidadesSuma = [[item[0], item[1]/n] for item in recuentoValoresSuma]
print("probabilidadesSuma=", probabilidadesSuma)
```

```
n= 36
probabilidadesSuma= [[2, 0.02777777777777776], [3, 0.05555555555555555], [4,
0.0833333333333333], [5, 0.1111111111111111], [6, 0.1388888888888889], [7,
0.1666666666666666], [8, 0.1388888888888889], [9, 0.1111111111111111], [10,
0.0833333333333333], [11, 0.0555555555555555], [12, 0.02777777777777776]]
```

Estos resultados son (las versiones numéricas de) los mismos que aparecen en la Tabla 4.1 del libro (pág. 97). Por ejemplo, la probabilidad correspondiente al valor 5 es  $\frac{4}{36}$  que es, aproximadamente:

```
print(4/36)
```

```
0.1111111111111111
```

Y es muy importante no perder de vista que, en tanto que probabilidades, se trata de valores teóricos. Lo que vamos a hacer a continuación es una simulación del experimento que consiste en lanzar dos dados y sumarlos, para comparar las frecuencias (empíricas o experimentales) que obtenemos con esas probabilidades (teóricas) que predice la variable  $X$ .

### Ejercicio 1.

*Este ejercicio debería resultar sencillo, después del trabajo de los tutoriales previos. Lo que queremos es simular  $n = 1000000$  tiradas de dos dados, y calcular la tabla de frecuencias relativas de la variable*

$$X = \{\text{suma de los dos dados}\}.$$

*Solución en la página 13.*



## 1.2. Media, varianza y desviación típica.

En este apartado vamos a ocuparnos de los cálculos necesarios para trabajar con una variable aleatoria discreta  $X$ , dada mediante una tabla de valores y probabilidades (la tabla de densidad de probabilidad) como esta:

Valor	$x_1$	$x_2$	$\dots$	$x_k$
Probabilidad	$p_1$	$p_2$	$\dots$	$p_k$

La teoría correspondiente se encuentra en el Capítulo 4 del libro. A partir de la información de esta tabla, queremos calcular la media  $\mu_X$  de  $X$  y la varianza  $\sigma_X^2$  de  $X$ . Vamos a aprender a utilizar Python para calcularlos.

Para fijar ideas vamos a pensar en un ejemplo concreto. Supongamos que la densidad de probabilidad de la variable  $X$  es esta:

Valor	2	4	7	8	11
Probabilidad	1/5	1/10	1/10	2/5	1/5

Vamos a almacenar los valores y las probabilidades, en una lista de pares. El primer elemento de cada par es el valor y el segundo la probabilidad de ese valor:

```
# Definicion de la variable X.  
X = [[2, 1/5], [4, 1/10], [7, 1/10], [8, 2/5], [11, 1/5]]
```

Y ahora, para calcular la media, haremos:

```
media = sum([x[0] * x[1] for x in X])  
print("Media de X = {:.4f}".format(media))
```

```
Media de X = 6.9000
```

mientras que la varianza y desviación típica se obtienen importando el módulo `math`:

```
import math as m
```

haciendo

```
varianza = sum([(x[0] - media)**2 * x[1] for x in X])  
print("varianza = {:.4f}".format(varianza))
```

```
varianza = 9.4900
```

y después:

```
sigma = m.sqrt(varianza)  
print("desviacion tipica = {:.4f}".format(sigma))
```

```
desviacion tipica = 3.0806
```

### Ejercicio 2.

1. Comprueba, usando Python, los resultados de los Ejemplos 4.2.2 y 4.2.6 del libro (págs. 105 y 108, respectivamente), en lo que se refiere a la variable  $X$ , suma de dos dados.
2. En el apartado anterior habrás obtenido un valor numérico (aproximado) de la varianza de  $X$ . Usa un programa simbólico (Wiris o Wolfram Alpha, por ejemplo) para calcular el valor exacto de la varianza.
3. Repite los apartados anteriores para la variable  $Y$ , la diferencia (en valor absoluto) de los dos dados.

Solución en la página 13. □

### 1.3. Operaciones con variables aleatorias.

En el Ejercicio 2 acabamos de calcular la media y varianza de las variables  $X$  e  $Y$ , que representan la suma y diferencia de los resultados al lanzar dos dados, respectivamente. Vamos a usar estas dos variables para experimentar con los resultados teóricos que aparecen en la Sección 4.3 del libro (pág. 109).

Es importante recordar siempre que las variables aleatorias son modelos *teóricos* de las asignaciones de probabilidad. La media  $\mu_X$  de la variable aleatoria  $X$  representa el valor medio esperado en una serie muy larga de repeticiones del suceso aleatorio que representa la variable  $X$ . Por tanto, la media sirve para hacer predicciones *teóricas* y, en cada caso concreto, los valores que obtendremos serán *parecidos, pero no idénticos* al valor que predice la media.

Para ilustrar esto, vamos a tomar como punto de partida la variable  $X$  (suma al lanzar dos dados), y definiremos una nueva variable:

$$W = 3 \cdot X - 4.$$

La teoría predice que ha de ser:

$$E(W) = E(3 \cdot X - 4) = 3 \cdot E(X) - 4$$

y, usando los resultados del Ejercicio 2 de este tutorial, tenemos

$$E(W) = 3 \cdot E(X) - 4 = 3 \cdot 7 - 4 = 17.$$

Para “*comprobar experimentalmente*” esta predicción teórica vamos a fabricar una serie muy larga ( $n = 10000$ ) de valores aleatorios de  $W$ , y calcularemos su media. Los valores de  $W$  se obtienen de los de  $X$  con este código Python (la primera parte es muy parecida al comienzo de la solución del Ejercicio 1):

```
import random as rnd
rnd.seed(2016)
n = 10000
dado1 = [rnd.randrange(1, 7) for _ in range(0, n)]
dado2 = [rnd.randrange(1, 7) for _ in range(0, n)]
X = [dado1[_] + dado2[_] for _ in range(0, n)]
W = [3 * x - 4 for x in X]
```

La novedad, naturalmente, es esa última línea, en la que calculamos los valores de  $W$  a partir de los de  $X$ . La media de esos 10000 valores de  $W$  es:

```
import numpy as np
mediaW = np.mean(W)
print("media de W= {:.4f}".format(mediaW))
```

```
media de W= 16.9853
```

Hemos usado la función `mean` de `numpy` para obtener el resultado. Y, como puedes ver, el resultado del experimento se parece mucho a nuestra predicción teórica.

Vamos a aprovechar, también, para comprobar que las cosas no siempre son tan sencillas. En particular, vamos a usar la variable

$$V = X^2$$

para comprobar que:

$$E(V) = E(X^2) \neq (E(X))^2 = 7^2 = 49.$$

Aquí, de nuevo,  $X$  es la variable suma al lanzar dos dados. Para comprobar *experimentalmente* esto procedemos de forma muy parecida a lo que acabamos de hacer con  $W$ . Generamos una lista de valores de  $V$ , y calculamos su media:

```
V = [x**2 for x in X]
mediaV = np.mean(V)
print("media de V= {:.4f}".format(mediaV))
```

```
media de V= 54.8099
```

¿Cuál es el cálculo teórico correspondiente? Para calcular la media de  $V = X^2$  empezamos por hacer la tabla de densidad de esta variable. Esa tabla se obtiene fácilmente de la Tabla 4.2.2 del libro (pág. 105), elevando los valores al cuadrado (las probabilidades se mantienen):

Valor	4	9	16	25	36	49	64	81	100	121	144
Probabilidad	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

Y ahora puedes usar cualquier programa (Wolfram Alpha, o el propio R) para comprobar que:

$$\mu_V = \frac{1974}{36} \approx 54.83.$$

Fíjate en que este valor se parece mucho más al que hemos obtenido en la versión experimental del cálculo.

Los resultados que acabamos de obtener confirman que la media no se lleva bien con el cuadrado: la media del cuadrado no es el “cuadrado de la media”. De hecho, la diferencia entre esas dos cantidades es, precisamente, la varianza:

$$\text{Var}(X) = E(X^2) - (E(X))^2.$$

Sin entrar a dar una demostración teórica de este hecho, vamos a comprobarlo usando la variable  $X$ . Empezamos repitiendo los mismos cálculos que aparecen en la solución del Ejercicio 2 (ver página 13).

```
valoresX = range(2, 13)
probabilidadesX = list(range(1,7)) + list(range(5, 0, -1))
probabilidadesX = [p/36 for p in probabilidadesX]
muX = sum([valoresX[i] * probabilidadesX[i] for i in range(0, len(valoresX))])
print("Media de X = {0:0.4f}".format(muX))
varX = sum([(valoresX[i] - muX)**2 * probabilidadesX[i] for i in range(0, len(valoresX))])
print("Varianza de X = {0:0.4f}".format(varX))
```

```
Media de X = 7.0000
```

```
Varianza de X = 5.8333
```

Recuerda que el cálculo que estamos haciendo aquí es teórico (no es un “experimento”). Ahora vamos a calcular la media de  $V = X^2$ :

```
valoresV = [x**2 for x in valoresX]
probabilidadesV = probabilidadesX
muV = sum([valoresV[i] * probabilidadesV[i] for i in range(0, len(valoresV))])
print("Media de V = {0:0.4f}".format(muV))
```

```
Media de V = 54.8333
```

Y ahora podemos comprobar, en este ejemplo, la identidad  $\text{Var}(X) = E(X^2) - (E(X))^2$ . Se tiene:

```
print("varX ={0:0.4f}".format(varX))
print("muV - (muX)**2 ={0:0.4f}".format(muV - (muX)**2))
```

```
varX =5.8333
muV - (muX)**2 =5.8333
```

como esperábamos. Naturalmente, este resultado teórico también se puede comprobar experimentalmente. Y es interesante hacerlo, así que lo exploraremos en los ejercicios adicionales.

## 1.4. Función de distribución (probabilidad acumulada)

La función de distribución de la variable aleatoria  $X$  es, recordémoslo:

$$F(k) = P(X \leq k)$$

Es decir, que dado el valor de  $k$ , debemos sumar todos los valores de la densidad de probabilidad para valores menores o iguales que  $k$ . En el ejemplo de la variable  $X$  que aparece al comienzo de la Sección 1.2 (pág. 3), si queremos calcular  $F(7)$ , debemos hacer:

$$F(7) = P(X = 2) + P(X = 4) + P(X = 7) = \frac{1}{5} + \frac{1}{10} + \frac{1}{10}.$$

Se trata de sumas acumuladas, como las que vimos en el caso de las tabla de frecuencias acumuladas. Así que usaremos lo que aprendimos en el Tutorial02 (Sección ??). Vamos a empezar por extraer las probabilidades de la variable  $X$ :

```
X = [[2, 1/5], [4, 1/10], [7, 1/10], [8, 2/5], [11, 1/5]]
valoresX = [item[0] for item in X]
probabilidadesX = [item[1] for item in X]
print(probabilidadesX)
```

```
[0.2, 0.1, 0.1, 0.4, 0.2]
```

Y ahora aplicamos la función `cumsum` de `numpy` así:

```
print(FdistX)
```

```
[0.2, 0.3000000000000004, 0.4, 0.8, 1.0]
```

que, como ves, produce un vector con los valores de  $F(k)$  para cada  $k$ . Seguramente preferiremos ver estos resultados en forma de tabla, para poder localizar fácilmente el valor de  $F$  que corresponde a cada valor de  $k$ . Con lo que hemos aprendido sobre la función `print`, es fácil conseguirlo. Pondrámos:

```
k = len(valoresX)
print("\nTabla de la variable aleatoria X:\n")
linea = "_" * 49
print(linea)
print("| Valor x | Probabilidad p | Fun. de dist. F(x) |")
print(linea)
for i in range(0, k):
    print("| {0: 7d} | {1: 14.2f} | {2: 18.2f} |".format(valoresX[i], \
        probabilidadesX[i], FdistX[i]))
print(linea)
```

Tabla de la variable aleatoria X:

Valor x	Probabilidad p	Fun. de dist. F(x)
2	0.20	0.20
4	0.10	0.30
7	0.10	0.40
8	0.40	0.80
11	0.20	1.00

Aunque en esta tabla sólo aparecen los valores 2, 4, 7, 8 y 11, es bueno recordar que la función de distribución  $F(x)$  está definida *sea cual sea el valor de x*. Es decir, que tiene perfecto sentido preguntar, por ejemplo, cuánto vale  $F(\frac{8}{3})$ . Más adelante vermeos la forma de conseguir que Python conteste esta pregunta automáticamente, pero todavía tenemos que aprender algunas cosas más sobre el lenguaje antes de ver cómo podemos conseguir eso.

### Ejercicio 3.

¿Cuánto vale  $F\left(\frac{8}{3}\right)$ ? Aunque no forme parte del ejercicio, trata de ir pensando en un procedimiento que te permita, dado un valor  $x$  cualquiera, obtener el valor  $F(x)$ . Solución en la página 15.  $\square$

## 1.5. Representación gráfica de las variables aleatorias.

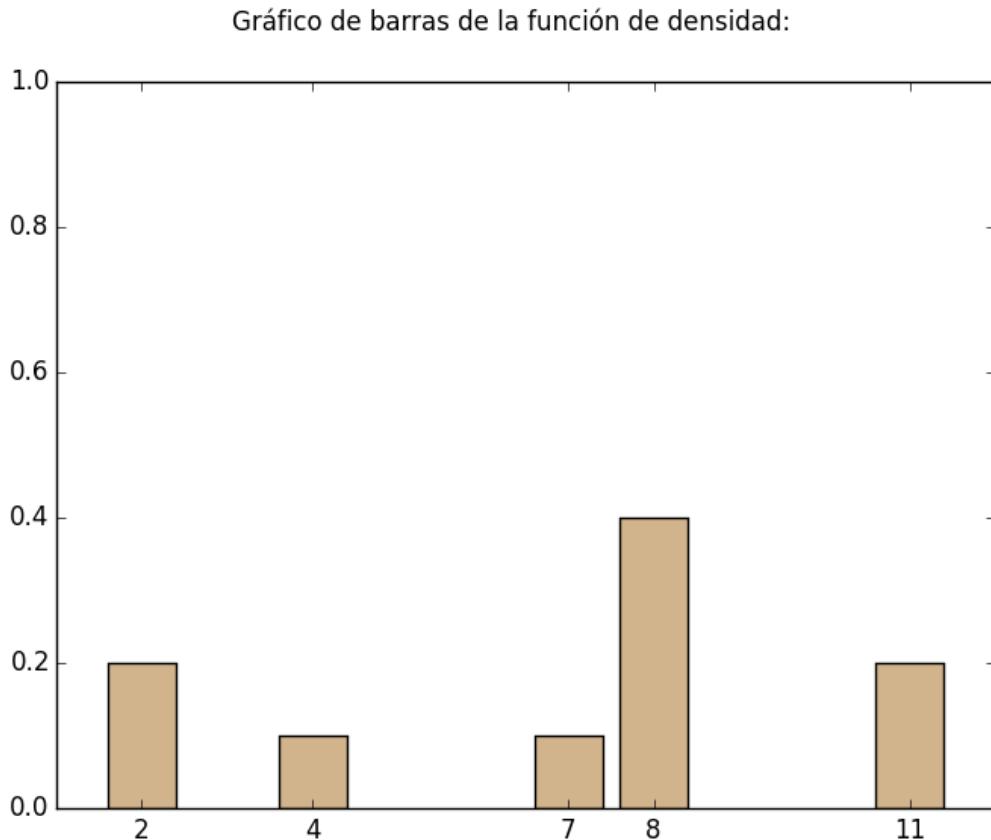
Dada una variable aleatoria  $X$ , por ejemplo la que venimos usando desde el comienzo de la Sección 1.2, podemos representar gráficamente su tabla de densidad de probabilidad, en un diagrama de columnas, usando la función `bar` de `matplotlib` (que ya encontramos en el Tutorial02). Empezamos importando el módulo:

```
import matplotlib.pyplot as plt
```

Y luego usaremos estos comandos:

```
plt.suptitle("Gráfico de barras de la función de densidad:")
plt.xticks(valoresX)
plt.axis([min(valoresX) - 1,max(valoresX) + 1, 0, 1])
plt.bar(valoresX, probabilidadesX, color='tan', align='center')
```

El resultado es esta figura:



Algunos comentarios:

- La función `suptitle` añade un título al gráfico.
- La función `xticks` nos sirve para indicarle a Python donde queremos que vayan situadas las etiquetas del eje  $x$ . En relación con esto, en la función `bar` hemos usado la opción `align='center'` para situar las etiquetas en el centro de cada columna (y no al principio). Esta era una tarea que habíamos dejado pendiente en el Tutorial02.

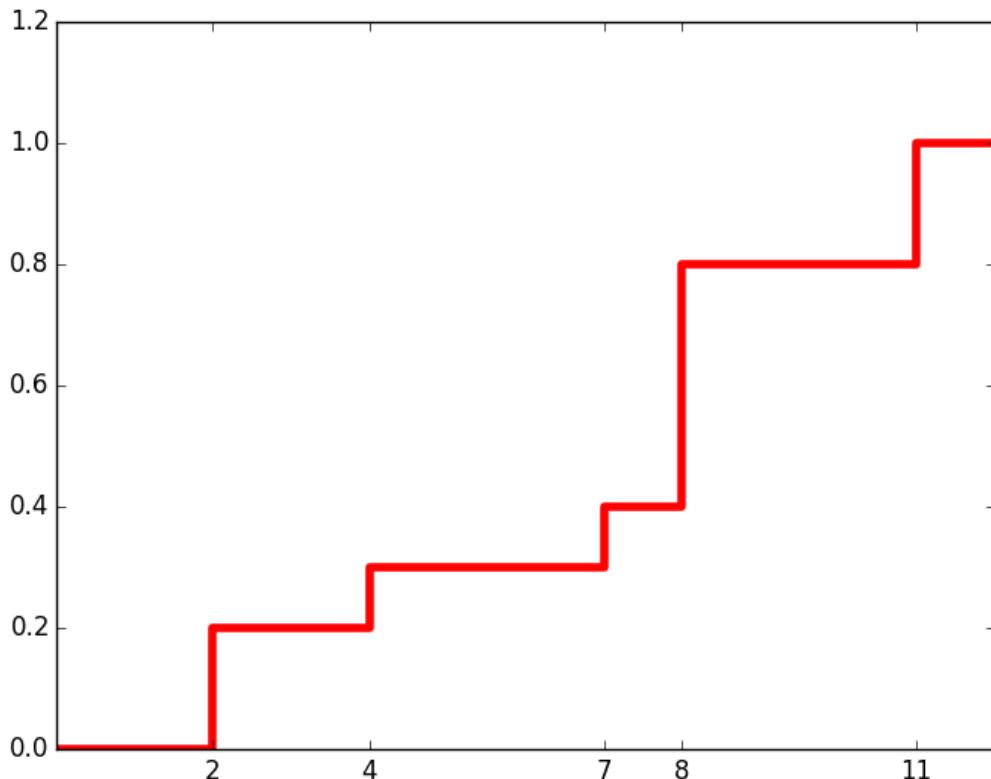
- La función `axis` sirva para fijar la `em` ventana gráfica que Python usará en la figura. Lo hacemos mediante una lista de cuatro valores que definen los valores mínimo y máximo, primero del eje  $x$  y luego del eje  $y$ . Para este ejemplo concreto nos hemos asegurado de que el eje  $x$  cubra todo el rango de valores de la variable  $X$ , con un margen de una unidad extra por cada lado, y que el eje  $y$  recorra los valores del 0 al 1, puesto que se trata de probabilidades.

Para representar la función de distribución es más común utilizar *gráficos de escalera* como el que aparece en la Figura 4.3 del libro (página 114). En Python hay varias maneras de hacerlo, más o menos complicadas. Aquí vamos a ver una bastante sencilla, que usa la función `step` (en el sentido inglés de *peldaño*). Como verás, en esa función hemos hecho algunos retoques, añadiendo en el eje  $x$  un valor a la izquierda del recorrido de  $X$  y uno a su derecha, que se corresponden con los valores 0 y 1.00001 del eje  $y$ . Lo hemos hecho para obligar a Python a crear una perspectiva algo más amplia de la función de distribución.

```
plt.suptitle("Gráfico de escalera de la función de distribucion:")
plt.xticks(valoresX)
plt.step([min(valoresX) - 2] + valoresX + [max(valoresX) + 1],
         [0] + FdistX + [1.00001], where='post', linewidth=4.0, color='red')
```

El resultado es esta figura:

Gráfico de escalera de la función de distribucion:



que, si bien dista de ser perfecta, es suficiente mientras recuerdes que las funciones de distribución son *continuas por la derecha*; en términos más sencillos, que los puntos gordos de la Figura 4.3 del libro (pág. 114) están en el extremo izquierdo de los peldaños.

## 1.6. Un fichero de comandos Python para estudiar una variable discreta.

Al igual que hicimos en el Tutorial02, en el que incluimos un fichero que resumía muchos comandos de Estadística Descriptiva, vamos a incluir aquí un *fichero plantilla* que reúne los comandos que hemos ido viendo en este Tutorial para trabajar con una variable aleatoria discreta (con un número finito de valores) definida mediante su tabla de densidad:

[Tut04-VariableAleatoriaDiscreta.py](#)

cuyo listado es:

```
###  
www.postdata-statistics.com  
POSTDATA. Introducción a la Estadística  
Tutorial 04.  
Fichero de comandos Python para el estudio de  
una variable aleatoria discreta.  
## Importacion de Modulos  
  
import numpy as np  
import matplotlib.pyplot as plt  
import math as m  
  
# Definicion de X a partir de valores y probabilidades.  
  
valoresX = [2, 4, 7, 8, 11]  
probabilidadesX = [1/5, 1/10, 1/10, 2/5, 1/5]  
X = [[valoresX[_], probabilidadesX[_]] for _ in range(0, len(valoresX))]  
  
# Alternativamente, definicion de la variable X como lista de pares [valor, probabilidad].  
# Descomentar la siguiente linea para usarla.  
  
# X = [[2, 1/5], [4, 1/10], [7, 1/10], [8, 2/5], [11, 1/5]]  
  
# En cualquier caso:  
  
valoresX = [x[0] for x in X]  
probabilidadesX = [x[1] for x in X]  
  
# Calculo de la media.  
  
media = sum([x[0] * x[1] for x in X])  
print("Media de X = {0:0.4f}".format(media))  
  
# Calculo de la varianza y desviacion tipica.  
  
varianza = sum([(x[0] - media)**2 * x[1] for x in X])  
print("varianza = {0:0.4f}".format(varianza))  
  
sigma = m.sqrt(varianza)  
print("desviacion tipica = {0:0.4f}".format(sigma))  
  
# Función de distribucion.  
  
FdistX = np.cumsum(probabilidadesX).tolist()  
  
# y su tabla:  
  
k = len(valoresX)
```

```

print("\nTabla de densidad de la variable aleatoria X:\n")
linea = "_" * 49
print(linea)
print("| Valor x | Probabilidad p | Fun. de dist. F(x) |")
print(linea)
for i in range(0, k):
    print("| {0: 7d} | {1: 14.2f} | {2: 18.2f} |".format(valoresX[i], \
        probabilidadesX[i], FdistX[i]))
print(linea)

# Gráfico de barras de la función de densidad.

plt.suptitle("Gráfico de barras de la función de densidad:")
plt.xticks(valoresX)
plt.axis([min(valoresX) - 1, max(valoresX) + 1, 0, 1])
plt.bar(valoresX, probabilidadesX, color='tan', align='center')

# Reset gráfico.

plt.figure()

# Gráfico de escalera de la función de distribucion.

plt.suptitle("Gráfico de escalera de la función de distribucion:")
plt.xticks(valoresX)
plt.step([min(valoresX) - 2] + valoresX + [max(valoresX) + 1],
         [0] + FdistX + [1.00001], where='post', linewidth=4.0, color='red')

```

## 2. Funciones definidas por el usuario en Python.

Opcional: aunque esta sección puede omitirse en una primera lectura, pronto se hará necesaria.

En esta sección vamos a aprender a escribir nuestras propias funciones Python. Antes de discutir más a fondo sobre el uso de las funciones empezaremos por ver algunos ejemplos muy sencillos. De esa forma confíamos en que te resulte más fácil entender la discusión sobre la necesidad y conveniencia de las funciones.

Vamos por tanto con el primero de esos ejemplos, que va a ser sencillo porque de momento queremos centrarnos en la forma de escribir una función. En concreto, vamos a escribir una función de Python, a la que llamaremos esCuadrado y que, dado un número entero  $n$ , nos diga si ese número es un cuadrado perfecto. La respuesta será un valor booleano, **True** or **False**, según que el número sea o no un cuadrado perfecto. Por ejemplo, queremos que al ejecutar:

```
esCuadrado(9)
```

la respuesta sea **True**, porque  $9 = 3^2$ , mientras que al ejecutar

```
esCuadrado(7)
```

queremos que la respuesta sea **False**.

Una función es, un cierto sentido, como un programa dentro de nuestro programa. Así que para diseñar la función empezamos usando pseudocódigo, como hacíamos con los programas. En este caso, por ejemplo, el plan es este:

1. Calcular la raíz cuadrada de  $n$  (que será un número real, no necesariamente entero).
2. Redondearla al entero más cercano.

3. Elevar ese entero al cuadrado y comprobar si coincide con \$n\$.
4. Si coincide responder True, en caso contrario responder False.

Usando ese pseudocódigo como referencia, crear la función es muy sencillo. Primero nos aseguramos de haber importado el módulo `math`:

```
import math as m
```

y ahora vamos con la función propiamente dicha:

```
def esCuadrado(n):
    """
    Devuelve True si el entero n es un cuadrado perfecto y False en caso contrario.
    """
    raiz = m.sqrt(n)
    raizRedondeada = round(raiz)
    if(raizRedondeada**2 == n):
        return(True)
    else:
        return(False)
```

Enseguida vamos a analizar detenidamente este código. Pero antes, veamos cómo funciona en el par de casos que antes hemos propuesto:

```
print(esCuadrado(9))
```

```
True
```

Y de modo análogo:

```
print(esCuadrado(7))
```

```
False
```

Como ves, la función que hemos creado se usa como cualquier otra función de Python. Vamos con el análisis del código de la función.

1. La primera línea, la **cabecera** de la función, comienza con la palabra clave `def`. Esa palabra sirve para avisar a Python de que comienza la definición de una función. A continuación escribimos el nombre de la función `esCuadrado` y, entre paréntesis, el argumento (o argumentos, como veremos pronto) de la función, que en este caso es el número `n`. La línea de cabecera termina con dos puntos que, como ya vamos reconociendo, es la forma de indicar en Python que comienza un *bloque* de instrucciones.
2. Las siguientes líneas indentadas forman lo que denominamos el **cuerpo** de la función. Python detecta el final de la función cuando desaparece esa indentación y volvemos al nivel de la línea de cabecera. Si escribes funciones en un buen editor de texto, que reconozca la sintaxis de Python, te resultará más fácil adaptarte a esta escritura de las funciones, porque el editor se encargará de forma automática de dar formato a lo que escribes.
3. Las primeras líneas del cuerpo de la función, delimitadas por las dos líneas que contienen tres comillas dobles `"""` forman un **bloque de documentación** inicial de la función. Nos hemos encontrado ya con este tipo de bloques de comentarios que ocupan varias líneas en las cabeceras de nuestros ficheros plantilla. Y al igual que sucede con el otro tipo de comentarios que ya conocemos (y que usan `#`), cuando Python encuentra estas líneas al principio del código de una función simplemente las ignora. De esa forma disponemos de un espacio en el que explicar qué es lo que hace la función. Como ocurre casi siempre con la documentación del código, no es en absoluto necesario que exista este bloque para que la función haga su trabajo correctamente. Pero hemos querido desde el primer ejemplo incluir la documentación como parte esencial de la escritura de la función, porque como repetiremos varias veces a lo largo del curso, **el código mal documentado es una mala práctica**. Más adelante volveremos sobre estas líneas iniciales de la función y sobre las diferencias entre usar `"""` y usar `#`.

- Como puedes ver, las restantes líneas del cuerpo de la función son simplemente instrucciones Python que ya conocemos y que traducen los pasos que hemos enumerado antes en el pseudocódigo. El cuerpo de la función incluye, al final, un bloque `if/else`. Hemos elegido este ejemplo para ilustrar el hecho de que el cuerpo de una función puede contener muchos de los elementos que hemos ido conociendo del lenguaje Python: asignaciones, bucles `for`, sentencias `if/else`, como en este ejemplo. Además la sentencia sentencias `if/else` de nuestro ejemplo contiene otro ingrediente fundamental de una función en Python: la función `return`.
- Toda función Python debería incluir al menos una llamada a la función `return`. El argumento de esa función define el valor que la función devuelve cuando la invocamos. En este ejemplo, como ves, tenemos dos apariciones de `return`. En la primera el valor que devuelve la función `esCuadrado` es `True`, y en la segunda es `False`. Fijate en que podríamos haber escrito la función de manera que sólo hubiera una aparición de `return`. Por ejemplo, así:

```
def esCuadrado(n):
    """
    Devuelve True si el entero n es un cuadrado perfecto y False en caso contrario.
    """
    raiz = m.sqrt(n)
    raizRedondeada = round(raiz)
    if(raizRedondeada**2 == n):
        respuesta = True
    else:
        respuesta = False
    return(respuesta)
```

Pero a veces es más natural usar varias veces `return`. En algunas ocasiones nos encontraremos con funciones en las que no es necesario definir un resultado de salida. Por ejemplo, funciones que producen un objeto externo como un fichero de datos o una figura. En esos casos se puede usar la función `return` sin argumentos, así:

```
return()
```

De esa forma simplemente le indicamos a Python que la ejecución de la función ha terminado. Cuando aprendas más sobre Python descubrirás que, en cualquier caso, siempre suele ser buena idea que la función produzca algún valor de salida. Por ejemplo, si la función crea un fichero de datos, el valor de salida puede ser un código que nos permita saber si el proceso de creación del fichero ha tenido éxito o si, por el contrario, se ha producido un problema y de qué tipo.

En cualquier caso, es importante saber que, tras ejecutar `return`, Python considera terminada la ejecución de la función y devuelve el control al punto de nuestro programa desde el que se invocó a la función. También es necesario saber que el valor que devuelve la función puede ser cualquiera de los objetos Python que hemos ido encontrando: números, booleanos o cadenas de caracteres, desde luego. Pero también listas, tuplas, etc. Incluso podemos tener funciones que produzcan como resultado otra función. En estos tutoriales tendremos ocasión de encontrarnos con algunas funciones más complejas.

### ¿Para qué sirve escribir nuestras propias funciones?

Desde el comienzo de nuestro trabajo con Python hemos ido aumentando la colección de funciones del lenguaje que conocemos. Las funciones son un ingrediente esencial de cualquier lenguaje de programación moderno. Y Python cuenta con una colección extensísima de funciones, especialmente gracias a la enorme cantidad de módulos que podemos importar. ¿Por qué son tan importantes las funciones en Programación? En su libro *Think Python* (ver la referencia [1] al final del tutorial), Allen B. Downey cita varias razones, que en esencia son estas:

- Escribir una función hace que nuestro código sea más fácil de escribir y leer. Sólo por eso ya merecerían la pena.
- Relacionado con lo anterior, las funciones simplifican enormemente el mantenimiento del código. Una máxima que conviene recordar es que el tiempo más valioso no es normalmente el tiempo que el ordenador pasa ejecutando el programa, sino el tiempo que el programador pasa escribiéndolo y corrigiéndolo.

- Desde el punto de vista metodológico, estructurar un programa usando funciones nos permita aplicar una estrategia *divide y vencerás* al desarrollo de los programas.
- A menudo descubriremos que una misma función se puede utilizar en muchos programas. Ya has visto ejemplos: todas las funciones que importamos desde los módulos `math` o `random`, etc. han sido escritas (y son actualizadas) por programadores del equipo de desarrollo de Python, pero todos los demás usuarios nos beneficiamos de ellas. De esa forma, el código agrupado en una función puede reciclarse y compartirse.

Nos gustaría detenernos en este último punto. Es conveniente recordar, cada vez que usamos las funciones de Python, que nuestro trabajo depende y se beneficia del esfuerzo previo de muchos otros programadores. Tal vez dentro de un tiempo llegues a escribir funciones tan interesantes que puedas compartirlas con la comunidad de programadores y de esa forma contribuir a esta tarea colectiva.

## Soluciones de algunos ejercicios

- Ejercicio 1, pág. 2

```
# Importamos el módulo random e inicializamos el generador
# de números pseudoaleatorios.
import random as rnd
rnd.seed(2016)
# Elegimos el número de tiradas.
n = 10000
# Generamos los resultados de los dados.
dado1 = [rnd.randrange(1, 7) for _ in range(0, n)]
dado2 = [rnd.randrange(1, 7) for _ in range(0, n)]
# Las correspondientes sumas.
suma = [dado1[_] + dado2[_] for _ in range(0, n)]
# Ahora hacemos la tabla de frecuencias absolutas de las sumas.
import collections as cl
sumaCounter = cl.Counter(suma)
freqAbsolutaSuma = sumaCounter.most_common()
freqAbsolutaSuma.sort()
print(freqAbsolutaSuma)
# Y la tabla de frecuencias relativas:
freqRelativaSuma = [[item[0], item[1]/n] for item in freqAbsolutaSuma]
print("frecuencias relativas de la suma=")
print(freqRelativaSuma)
```

```
[(2, 280), (3, 582), (4, 858), (5, 1061), (6, 1381), (7, 1618), (8, 1450), (9, 1113), (10, 822), (11, 561), (12, 274)]
frecuencias relativas de la suma=
[[2, 0.028], [3, 0.0582], [4, 0.0858], [5, 0.1061], [6, 0.1381], [7, 0.1618], [8, 0.145], [9, 0.1113], [10, 0.0822], [11, 0.0561], [12, 0.0274]]
```

Recuerda comparar estas frecuencias relativas (experimentales) con las probabilidades (teóricas):

- Ejercicio 2, pág. 3

1. Suponemos que la tabla de densidad de la variable suma está almacenada en la lista de pares `probabilidadesSuma` que hemos obtenido al principio de la Sección 1.1.

```
print(probabilidadesSuma)
```

```
[[2, 0.027777777777777776], [3, 0.0555555555555555], [4, 0.0833333333333333], [5, 0.1111111111111111], [6, 0.1388888888888889], [7, 0.1666666666666666], [8, 0.1388888888888889], [9, 0.1111111111111111], [10, 0.0833333333333333], [11, 0.0555555555555555], [12, 0.02777777777777776]]
```

Entonces podemos hacer

```
X = probabilidadesSuma
```

y limitarnos a aplicar el código que hemos visto en este apartado:

```
media = sum([x[0] * x[1] for x in X])
print("Media de X = {0:0.4f}".format(media))
import math as m
varianza = sum([(x[0] - media)**2 * x[1] for x in X])
print("varianza = {0:0.4f}".format(varianza))
sigma = m.sqrt(varianza)
print("desviacion tipica = {0:0.4f}".format(sigma))
```

```
Media de X = 7.0000
varianza = 5.8333
desviacion tipica = 2.4152
```

2. Debes obtener el valor de la varianza igual a  $\frac{35}{6}$ , como se indica en el ejemplo.
3. Para obtener la diferencia hacemos:

```
diferencia = [abs(d1 - d2) for d1 in dado1 for d2 in dado2]
```

La función `abs` sirve para calcular el valor absoluto de la diferencia. Su tabla de frecuencias se obtiene con:

```
diferenciaCounter = cl.Counter(diferencia)
recuentoValoresDiferencia = diferenciaCounter.most_common()
recuentoValoresDiferencia.sort()
print(recuentoValoresDiferencia)
```

```
[(0, 6), (1, 10), (2, 8), (3, 6), (4, 4), (5, 2)]
```

Las convertimos en probabilidades con:

```
n = len(diferencia)
print("n=", n)
probabilidadesDiferencia = [[item[0], item[1]/n] for item in recuentoValoresDiferencia]
print("probabilidadesDiferencia=", probabilidadesDiferencia)
```

```
n= 36
probabilidadesDiferencia= [[0, 0.1666666666666666], [1, 0.2777777777777778], [2, 0.2222222222222222], [3, 0.1666666666666666], [4, 0.1111111111111111], [5, 0.0555555555555555]]
```

Y ahora podemos calcular la media, varianza y desviación típica con:

```
Y = probabilidadesDiferencia

media = sum([y[0] * y[1] for y in Y])
print("Media de Y = {0:0.4f}".format(media))
import math as m
varianza = sum([(y[0] - media)**2 * y[1] for y in Y])
print("varianza = {0:0.4f}".format(varianza))
sigma = m.sqrt(varianza)
print("desviacion tipica = {0:0.4f}".format(sigma))
```

```
Media de Y = 1.9444  
varianza = 2.0525  
desviacion tipica = 1.4326
```

En esta ocasión hemos llamado **Y** a la variable diferencia porque ya teníamos una variable **X** en el mismo ejercicio. Pero eso nos obliga a cambiar el código de cálculo de la media y la varianza, renombrando las variables. No es una buena práctica, porque en esos cambios se pueden introducir errores y porque duplicamos código innecesariamente. Habría sido mejor en todo caso hacer

```
X = probabilidadDiferencia
```

y después copiar exactamente las mismas líneas de código que usamos cuando **X** era la variable suma. Pero hemos hecho esto precisamente para brindarte la ocasión de reflexionar sobre esto. Incluso la segunda opción, siendo preferible, incluye código duplicado. La mejor solución será evidente una vez que hayamos aprendido a escribir funciones en Python, más adelante en este mismo tutorial.

- **Ejercicio 3, pág. 7**

El valor es  $F\left(\frac{8}{3}\right) = 0.2$ , porque se tiene  $2 < \frac{8}{3} < 4$ , así que

$$F\left(\frac{8}{3}\right) = P(X \leq \frac{8}{3}) = P(X \leq 2) = F(2) = 0.2$$

La gráfica de la función de distribución (pág. 8) puede ayudarte a entender este resultado.

## Referencias

- 
- [1] Allen Downey. *Think Python*. O'Reilly Media, 2nd edition, 2015. Ebook ISBN: 978-1-4919-3935-2

---

Fin del Tutorial-04. ¡Gracias por la atención!