PROGRAMACIÓN CON PYTHON.

PARTE I: FUNDAMENTOS DE PROGRAMACIÓN CON PYTHON.

1. INTRODUCCIÓN.

1.1. ¿QUÉ ES LA PROGRAMACIÓN?

La programación es la acción de proporcionar a un ordenador las instrucciones que debe realizar. Las instrucciones del programa generado podrían servir para decirle al ordenador que opere ciertos datos, que modifique un texto, que busque información en un archivo, o que se comunique con otro ordenador a través de Internet.

Todos los programas usan instrucciones básicas como bloques constitutivos. Algunos ejemplos (en inglés) son los siguientes:

- **"Do this; then do that"**. (Haz esto; después, haz esto otro).
- **"If this condition is true, perform this action; oterwise, do that action"**. (Si esta condición es cierta, realiza esta acción; en caso contrario, haz esta otra acción).
- **"Do this action that number of times"**. (Haz esta acción este número de veces).
- **"Keep doing that until this condition is true"**. (Sigue haciendo esto hasta que esta condición se cumpla).

También podemos combinar estos bloques constitutivos para implementar decisiones más complejas. Por ejemplo, la figura muestra las instrucciones (el llamado **código fuente**) de un sencillo programa escrito en el lenguaje de programación Python. Comenzando desde arriba, el software de Python ejecuta secuencialmente cada línea del código (algunas líneas solo se ejecutan *si* una cierta condición es cierta) hasta que llega al final del programa.

```
❶ passwordFile = open('SecretPasswordFile.txt')
❷ secretPassword = passwordFile.read()
❸ print('Enter your password.')
  typedPassword = input()
❹ if typedPassword == secretPassword:
❺     print('Access granted.')
❻     if typedPassword == '12345':
❼         print ('That password is one that an idiot puts on their luggage.')
      else:
❽         print('Access denied.')
```

Sin saber nada de programación es probable que, simplemente leyendo el programa, puedas hacerte una idea de lo que hace. (1) En primer lugar, el programa abre al archivo "SecretPasswordFile.txt", y a continuación, (2) lee la contraseña secreta que contiene. (3) Después, le pide al usuario que introduzca una contraseña (mediante el teclado). En (4) y (5), el programa compara estas dos contraseñas, y si son iguales, muestra por pantalla el mensaje "Access granted" (acceso concedido). Luego, en (6), el programa comprueba si la contraseña proporcionada por el usuario es "12345", y en (7) indica que dicha elección no sería una buena opción para una contraseña. Por último, en (8), si las contraseñas no son iguales, el programa muestra por pantalla el mensaje "Access denied" (acceso denegado).

1.2. ¿QUÉ ES PYTHON?

Al hablar de **Python**, nos estamos refiriendo conjuntamente al lenguaje de programación Python (esto es, al conjunto de instrucciones y reglas sintácticas que se consideran válidas para escribir en un programa de Python), y al intérprete Python, que se encarga de leer ese código fuente (escrito en el lenguaje Python) y

de ejecutar sus instrucciones. El intérprete Python puede descargarse gratis de la web http://python.org/ en sus versiones para Windows, Linux, y OS X.[1]

1.3. DESCARGAR E INSTALAR PYTHON.

Podemos descargar Python gratuitamente de la web http://python.org/downloads/. Si descargas la última versión, todos los programas de este texto deberían funcionar correctamente.

> NOTA IMPORTANTE: Asegúrate de descargar una versión de Python 3 (como la versión 3.7.0). Los programas de este texto están escritos para funcionar en Python 3, y podrían no funcionar correctamente (o no funcionar en absoluto) en Python 2.

En esa web encontrarás los instalables de Python para las versiones de 64 bits y de 32 bits del sistema operativo correspondiente, así que lo primero que debes hacer es averiguar qué instalable necesitas. Si compraste tu ordenador en 2007 o después, lo más probable que tengas un sistema operativo de 64 bits. En caso contrario, tendrás una versión de 32 bits. Pero para asegurarte, haz lo siguiente:

- En Windows, elige "Comenzar" → "Panel de control" → "Sistema", y comprueba si el tipo de sistema es de 64 bits o de 32 bits.
- En OS X, ve al menú de Apple, elige "Acerca de este Mac" → "Más información" → "Informe de sistema" → "Hardware", y mira el campo "Nombre del procesador". Si pone Intel Core Solo o Intel Core Duo, tienes una máquina de 32 bits. Si pone cualquier otra cosa (incluyendo Intel Core 2 Duo), tiene una máquina de 64 bits.
- En Ubuntu Linux, abre un Terminal y ejecuta el comando "uname -m". La respuesta "i686" significa 2 bits, y la respuesta "x86_64" significa 64 bits.

Asumiendo que trabajamos en Windows, descarga el instalable (la extensión del archivo es .msi) y haz doble clic sobre él. A continuación, sigue las instrucciones que el instalable muestra por pantalla:

- Elige "Instalar para todos los usuarios" y clica "Siguiente".
- Instala en la carpeta "C:\Python34" clicando en "Siguiente".
- Clica en "Siguiente" una vez más para saltarte la opción de personalizar la instalación de Python.

1.4. ARRANCAR EL IDLE DE PYTHON.

El intérprete de Python es el software que permite ejecutar tus programas de Python. Pero para escribir los programas de Python, necesitamos usar el entorno de desarrollo interactivo o **IDLE** (Interactive DeveLopment Environment) de Python (el cual es muy parecido a un procesador de textos). Vamos a arrancarlo.

Asumiendo que trabajamos en Windows 7 o posteriores, clica en el botón de inicio en la esquina inferior izquierda de la pantalla, escribe IDLE en la casilla de búsqueda, y selecciona "IDLE (Python GUI)".

Al arrancar el IDLE de Python, e independientemente de nuestro sistema operativo, aparece una pantalla prácticamente en blanco, a excepción de un texto que debe parecerse a lo siguiente:

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

[1] Por cierto, que el nombre de Python proviene del grupo de cómicos británicos Monty Python, y no de la serpiente pitón.

Esta ventana se denomina **shell interactivo**. Un **shell** es un programa que nos permite escribir las instrucciones que debe ejecutar el ordenador, algo así como el terminal o la ventana de comandos de OS X y Windows, respectivamente. Mediante el shell interactivo de Python introducimos las instrucciones que ejecutará el intérprete de Python. El ordenador lee estas instrucciones y las ejecuta de forma inmediata.

Por ejemplo, escribe lo siguiente tras el "prompt" (>>>) del shell interactivo:

```
>>> print ('Hello, world.')
```

Después de escribir esta línea y presionar la tecla ENTER, el shell interactivo debería mostrar esta respuesta:

```
>>> print ('Hello, world.')
Hello, world.
```

1.5. CÓMO ENCONTRAR AYUDA.

Tú mismo puedes resolver problemas de programación de forma muy sencilla. Para ver cómo hacerlo, vamos a cometer un error a propósito: Escribe '42'+3 en el shell interactivo. Por el momento no necesitamos saber qué significa esta instrucción, pero el resultado debería ser algo parecido a esto:

```
>>> '42'+3
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    '42'+3
TypeError: can only concatenate str (not "int") to str
```

El mensaje de error en "TypeError" aparece porque Python no entiende la instrucción que has escrito. La parte "Traceback" del mensaje de error muestra la instrucción específica y el número de línea con el que Python ha tenido el problema. Introduce "TypeError: can only concatenate str (not "int") to str" (comillas incluidas) en tu buscador favorito, y encontrarás multitud de resultados explicando qué significa el mensaje de error, y qué lo ha podido provocar.

A menudo descubrirás que ha habido más gente que ha tenido la misma duda que tú previamente, y que otra persona ya se la solventó. Nadie puede saberlo todo sobre programación, por lo que una tarea muy habitual para un desarrollador de software es buscar respuestas a preguntas técnicas.

1.6. DÓNDE Y CÓMO HACER PREGUNTAS SOBRE PROGRAMACIÓN.

Si no has podido encontrar en Internet la respuesta a tu duda, intenta preguntar en foros web, como en el foro Stack Overflow (http://stackoverflow.com/), o en el subreddit "learn prgramming" en http://reddit.com/r/learnprogramming/. Pero ten en cuenta que debes plantear tus dudas de forma que queden claramente expresadas. Para ello, consulta las secciones de preguntas frecuentes (Frequently Asked Questions, FAQ) de estas webs para saber cuál es la forma adecuada de publicar una pregunta en el foro.

Al hacer una pregunta, recuerda hacer lo siguiente:

- Explica lo que estás intentando hacer.
- Especifica el punto exacto en el que ocurre el error.
- Copia y pega el mensaje de error completo y tu código en http://pastebin.com/ o en http://gist.github.com/. Estos sitios web permiten compartir grandes cantidades de código con otras personas en la web, sin el riesgo de perder el formato del texto. A continuación, puedes poner el enlace URL del código publicado en el foro.
- Explica lo que ya has intentado hacer para resolver el problema.

- Indica qué versión de Python estás usando, y el tipo y versión de tu sistema operativo.
- Si el error ocurrió nada más cambiar algo en tu programa, explica exactamente qué cambiaste.
- Indica si el error ocurre cada vez que se ejecuta el programa, o si solo ocurre después de realizar ciertas acciones. En su aso, explica cuáles son esas acciones.

1.7. ¿POR QUÉ PYTHON?

Llegados a este punto, puede que nos preguntemos por qué aprender Python, y no otro de entre los múltiples lenguajes de programación disponibles hoy en día. En ocasiones, la elección de un lenguaje de programación se basa en preferencias personales, pero los usuarios de Python coinciden en destacar las siguientes ventajas:

Sencillez: Python es un lenguaje diseñado para ser fácil de leer, y por consiguiente, muy reutilizable y fácilmente mantenible (mucho más que otros lenguajes tradicionales). La uniformidad del código Python lo hace muy fácil de entender, incluso para quien no lo sabe escribir.

Productividad: Python permite mejorar la productividad del desarrollador en comparación con otros lenguajes como C, C++, y Java. En efecto, un programa escrito en Python suele tener un tamaño equivalente a una tercera o a una quinta parte del tamaño que tendría el código equivalente en C++ o Java. Esto implica que en Python hay que escribir menos, hay que depurar menos, y hay que mantener menos. Además, los programas Python pueden ejecutarse inmediatamente, sin necesidad de pasar por las largas compilaciones previas que son necesarias en otros lenguajes.

Portabilidad: La mayoría de programas Python pueden ejecutarse en todas las plataformas informáticas sin necesidad de cambiar el código. Así, portar el código de Linux a Windows es una mera cuestión de copiar el programa.

Librerías: Python viene por defecto con una gran cantidad de funcionalidades preconstruidas, conocidas como la **librería estándar**. Además, Python puede ampliarse con librerías propias y una gran colección de aplicaciones y software de terceros (para construcción de sitios web, para el desarrollo de videojuegos, para programación numérica (aplicaciones matemáticas), etc.). Por ejemplo, la extensión NumPy es una potente (y gratuita) extensión para programación numérica.

Integración: Los programas Python se comunican fácilmente con otras partes de una aplicación, usando una gran variedad de mecanismos de integración. Hoy día, Python puede usar librerías propias de C y C++, puede ser llamado por programas escritos en C y C++, puede invocar componentes Java y .NET, se puede comunicar con plataformas como COM y Silverlight, puede interaccionar con dispositivos a través de puertos serie, e interatuar con redes mediante interfaces como SOAP, XML-RCP, y CORBA.

¿Cuál es su desventaja? Bueno, en comparación con otros lenguajes de programación, la velocidad de ejecución de los programs Python puede no ser tan buena como en los lenguajes compilados o de bajo nivel como C y C++. En algunas (muy contadas) aplicaciones, tal vez necesitemos "acercarnos más a la máquina" y usar lengiajes de más bajo nivel para mejorar la eficiencia.

1.8. ¿QUIÉN USA PYTHON HOY DÍA?

En el momento de escribir este texto, Python está entre los 5 o 10 lenguajes de programación más usados a nivel mundial, según la fuente consultada. Y como se ha estado usando durante más de dos décadas, es muy estable y robusto.

Además de ser ampliamente usado por usuarios individuales, Python también se emplea en compañías y organismos de gran relevancia:

- Google usa Python en sus sistemas de búsqueda en la web.
- El servicio de distribución de video YouTube está escrito en Python.
- El sercicio de almacenamiento Dropbox codifica su software de servidor y de cliente con Python.
- El sistema de intercambio de archivos BitTorrent comenzó como un programa escrito en Python.
- Industrial Light and Magic, Pixar, y otros usan Python en la producción de películas animadas.
- La NSA (la agencia de seguridad nacional estadounidense) usa Python para realizar sus análisis de inteligencia y criptografías.
- Netflix y Yelp han usado Python en el desarrollo de sus infraestructuras.
- Cisco, Intel, Hewlett-Packard, Seagate, Qualcomm, e IBM usan Python para testear su hardware.
- JPMorgan Chase, UBS, Getco, y Citadel aplican Python al análisis y la previsión de marcados financieros.
- La NASA, Los Alamos, Fermilab, JPL, y otros usan Python para tareas de programación científica.
- Etc.

1.9. ¿QUÉ SE PUEDE HACER CON PYTHON?

Por supuesto, Python es útil para realizar multitud de tareas del muldo real, esto es, el tipo de cosas que los desarrolladores software hacen en sus estudios y oficinas todos los días. Python se usa en una gran variedad de aplicaciones, y de hecho, como lenguaje de propósito general, las aplicaciones de Python son casi ilimitadas: Podemos usar Python para hacer desde desarrollo web hasta videojuegos y robótica.

Sin embargo, las aplicaciones más habituales de Python hoy día parecen encajar en unas pocas categorías, que listamos a continuación:

- Programación de sistemas.
- GUIs (interfaces gráficas de usuario).
- Programación en Internet.
- Integración de componentes.
- Programación de bases de datos.
- Prototipado.
- Programación numérica y científica.
- Y mucho más: videojuegos, procesado de imágenes, data-mining, robótica, Excel, etc.

2. CONCEPTOS BÁSICOS DE PYTHON.

El lenguaje de programación Python posee un amplio abanico de construcciones sintácticas, librerías de funciones estándar, y características interactivas de su entorno de desarrollo. Afortunadamente, para escribir programas básicos no necesitamos tener todos estos conocimientos. Sin embargo, sí que necesitamos aprender algunos conceptos básicos de programación antes de poder empezar a escribir programas. En este capítulo vamos a utilizar algunas instrucciones básicas de Python, escribiéndolas directamente en el shell interactivo, para ver qué resultado producen. Usar el shell interactivo es una muy buena forma de aprender para qué sirven las instrucciones básicas de Python.

2.1. INTRODUCIR EXPRESIONES EN EL SHELL INTERACTIVO.

Puedes abrir el shell interactivo ejecutando el IDLE de Python. En Windows, abre el menú de inicio, y selecciona "Todos los programas" → "Python" → "IDLE (Python GUI)". A continuación, debería aparecer una ventana con el "prompt" `>>>`. Ése es el shell interactivo. Ahora, introduce la expresión `2+2` para ver qué ocurre:

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2+2
4
>>>
```

En Python, `2+2` es una **expresión**, que es el tipo más básico de instrucción de programación del lenguaje. Las expresiones consisten en **valores** (como el `2`) y **operadores** (como el `+`), y siempre pueden **evaluarse** (esto es, reducirse) a un valor único. Esto significa que, en cualquier lugar donde Python te permita usar un valor único, también puedes usar una expresión en su lugar.

En el ejemplo previo, `2+2` se evalúa a un valor único, `4`. A un valor único sin operadores también se le considera una expresión, aunque este valor único se evalúa directamente sí mismo, como muestra la figura:

```
>>> 2
2
```

Python proporciona otros muchos operadores que podemos usar en las expresiones. La tabla muestra todos los operadores *matemáticos* disponibles en Python:

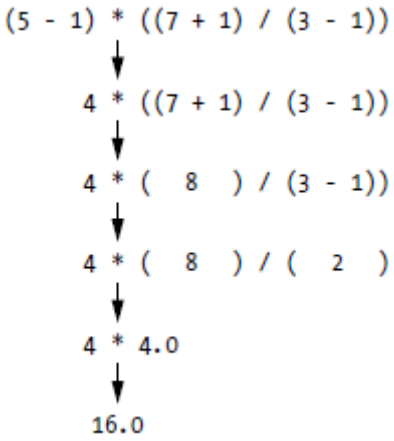
Operator	Operation	Example	Evaluates to...
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division/floored quotient	22 // 8	2
/	Division	22 / 8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	2 + 2	4

El orden en el que se hacen las diferentes operaciones combinadas en una expresión (la llamada **precedencia**) es similar a la de las matemáticas. El operador `**` se evalúa en primer lugar; a continuación se evalúan los operadores `*`, `/`, `//`, y `%`, de izquierda a derecha; en último lugar, se evalúan los operadores `+` y

– (también de izquierda a derecha). Podemos usar paréntesis para saltarnos la precedencia si así lo necesitamos. Escribe las siguientes expresiones en el shell interactivo de Python:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2      +           2
4
>>> (5 - 1)*((7 + 1)/(3 - 1))
16.0
```

En cada caso, tú debes introducir la expresión, pero es Python quien hace el trabajo difícil evaluándola a un valor único. Python seguirá evaluando cada parte de una expresión hasta haberla reducido a un valor único, como muestra la figura:



Estas reglas para combinar operadores y valores para formar expresiones son una parte fundamental del lenguaje de programación Python. Si escribimos incorrectamente una instrucción, Python será incapaz de entenderla, y mostrará un mensaje de error de tipo `SyntaxError`, como muestra la figura:

```
>>> 5+
SyntaxError: invalid syntax

>>> 45+5+*2
SyntaxError: invalid syntax
```

Para terminar recuerda que siempre es posible comprobar si una instrucción funciona, simplemente escribiéndola en el shell interactivo. No hay que preocuparse porque pueda ser incorrecta: lo peor que puede pasar es que Python responda con un mensaje de error.

2.2. LOS TIPOS DE DATOS ENTERO, PUNTO FLOTANTE, Y CADENA.

Recuerda que las expresiones solo son valores combinados mediante operadores, y que al final, siempre se reducen a un valor único. Un **tipo de datos** es la categoría a la que pertenece un valor, y cada valor solo

puede pertenecer a un tipo de datos. Los tipos de datos más comunes están listados en la tabla. Los valores como `-2`, `30`, etc. son ejemplos de valores **enteros**. El tipo de datos entero (integer) indica valores que son números enteros. Los números con un punto decimal, como `3.14` y `-5.05`, se denominan números de tipo **punto flotante** (floating - point). Notar que, aunque el valor `42` es un entero, el valor `42.0` sería un número de tipo punto flotante. Los números de tipo punto – flotante muy grandes o muy pequeños pueden expresarse usando notación científica, en la forma `1.2e34` (lo que significa $1{,}2 \times 10^{34}$) o `0.46e-19` (esto es, $0{,}46 \times 10^{-19}$),

Data type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

Los programas de Python también usan valores de tipo texto llamados **cadenas** (strings). Siempre debemos rodear un valor de tipo cadena mediante comillas simples (como por ejemplo, en `'Hola'` y en `'Adiós, mundo cruel!'`) para que Python sepa dónde empieza y dónde termina la cadena. Mediante la instrucción `''` también podemos tener una cadena sin caracteres dentro de ella, a la que se le llama **cadena vacía**. Si alguna vez obtienes el mensaje de error `SyntaxError: EOL while scanning string literal`, probablemente hayas olvidado poner una comilla simple al principio y/o al final de una cadena.

NOTA: Introduce la siguiente expresión en el shell interactivo de Python:

```
>>> 24/2
12.0
```

Observa que las expresiones con el operador `/` siempre se evalúan a un valor flotante. Además, las operaciones matemáticas que incluyan incluso un único flotante también se evalúan a valores flotantes. (Por ejemplo, `12.0+2` se evalúa a `14.0`).

2.3. CONCATENACIÓN Y RÉPLICA DE CADENAS.

El significado de un operador puede cambiar dependiendo del tipo de datos de los valores que opera. Por ejemplo, `+` es un operador de suma cuando opera sobre dos valores enteros o sobre dos valores de tipo punto flotante. Sin embargo, cuando lo usamos para operar dos valores de tipo cadena, el operador `+` actúa como un operador de **concatenación**, uniendo las dos cadenas. Inserta el siguiente código en el shell interactivo:

```
>>> 'Alice'+'Bob'
'AliceBob'
```

Esta expresión se evalúa a un valor único de tipo cadena que combina el texto de las dos cadenas. Pero si intentamos usar el operador `+` sobre una cadena y un entero, Python no sabrá cómo manejar esta situación, y mostrará un mensaje de error:

```
>>> 'Alice'+42
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    'Alice'+42
TypeError: can only concatenate str (not "int") to str
```

El mensaje `TypeError: can only concatenate str (not "int") to str` significa que Python solo puede concatenar una cadena a otra cadena, y no a un entero.

Ahora, el operador `*` se usa para multiplicar cuando operamos valores de tipo entero o de tipo punto flotante. Pero cuando `*` se usa sobre una cadena y un entero, se convierte en el operador de **replicación**. Inserta una cadena multiplicada por un número en el shell interactivo para ver este operador en acción:

```
>>> 'Alice'*5
'AliceAliceAliceAliceAlice'
```

La expresión se evalúa a un valor único que repite la cadena original un número de veces igual al valor entero.

El operador `*` solo puede usarse con dos valores numéricos (para la multiplicación), o bien con un valor tipo cadena y un valor entero (para la replicación de cadenas). En cualquier otro caso, Python mostrará un mensaje de error:

```
>>> 'Alice'*'Bob'
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    'Alice'*'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice'*5.0
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    'Alice'*5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Tiene sentido que Python no pueda entender estas dos expresiones: No podemos multiplicar dos palabras, y es muy difícil replicar una cadena arbitraria un número fraccionario de veces.

2.4. ALMACENAR VALORES EN VARIABLES.

Una **variable** es como una caja en la memoria del ordenador dentro de la cual podemos almacenar un valor único. Si más adelante en nuestro programa queremos usar el resultado evaluado de una expresión, podemos guardarlo en una variable.

SENTENCIAS DE ASIGNACIÓN.

Para almacenar valores en variables debemos usar una **sentencia de asignación**. Una sentencia de asignación consiste en un nombre de variable, un signo de igualdad (llamado **operador de asignación**), y el valor a almacenar. Si insertamos la sentencia de asignación `spam=42`, la variable llamada `spam` almacenará el valor entero `42` dentro de ella. Podemos imaginarnos la variable como una caja etiquetada en la que guardamos un cierto valor, como muestra la figura:

A modo de ejemplo, inserta el siguiente código en el shell interactivo:

```
❶ >>> spam = 40
  >>> spam
  40
  >>> eggs = 2
❷ >>> spam + eggs
  42
  >>> spam + eggs + spam
  82
❸ >>> spam = spam + 2
  >>> spam
  42
```

Una variable se **inicializa** (o se crea) la primera vez que almacenamos un valor dentro de ella, como en (1). Después de eso, podemos usarla en expresiones junto con otras variables o valores, como en (2). Cuando a una variable se le asigna un nuevo valor, como en (3), el valor antiguo se borra. Ésta es la razón por la que la variable `spam` se evalúa a `42` en vez de a `40` al final del ejemplo. A esto se le llama **sobrescribir** la variable. Introduce este código en el shell interactivo para sobrescribir una cadena:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

En este ejemplo, la variable `spam` almacena `'Hello'` hasta que reemplazamos este valor por `'Goodbye'`, tal y como ilustra la figura:

NOMBRES DE VARIABLES.

La tabla de la figura muestra algunos ejemplos de nombres de variables que son permisibles. Podemos nombrar una variable como prefiramos, siempre que el nombre obedezca estas tres reglas:

1) Solo puede ser una palabra.
2) Solo puede usar letras, números, y el carácter de barra baja (_).
3) No puede comenzar con un número.

Los nombres de variable distinguen entre mayúsculas y minúsculas, lo que significa que `spam`, `SPAM`, `Spam`, y `sPaM` son cuatro variables distintas. En Python, es tradición comenzar los nombres de variables con una letra minúscula.

Valid variable names	Invalid variable names
balance	current-balance (hyphens are not allowed)
currentBalance	current balance (spaces are not allowed)
current_balance	4account (can't begin with a number)
_spam	42 (can't begin with a number)
SPAM	total_$um (special characters like $ are not allowed)
account4	'hello' (special characters like ' are not allowed)

En este texto utilizaremos el estilo de escritura "camelCase" para nombrar a las variables, en lugar de separar las palabras mediante una barra baja. Así, nombraremos variables como `lookLikeThis`, y no como `look_like_this`. Algunos expertos podrían objetar que el código de estilo oficial de Python, PEP 8, dice que debemos usar barras bajas, pero también otorga al programador libertad total para aplicar una guía de estilos propia.

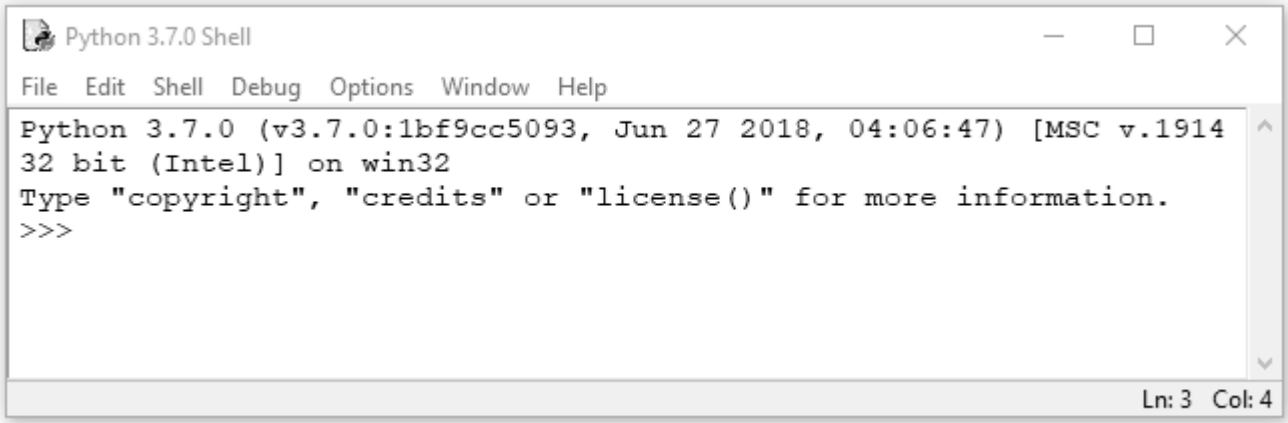
Por último, un buen nombre de variable debe describir adecuadamente el valor que contiene. Imagina que nos mudamos a una casa nueva, y que etiquetamos todas las cajas con el nombre `cosas`. Probablemente nunca encontraríamos nada. En este libro, y también en la documentación de Python, nos encontraremos con los nombres genéricos `spam`, `eggs`, y `bacon` para las variables de los ejemplos. Pero en nuestros programas, un nombre descriptivo hará que nuestro código sea mucho más legible.

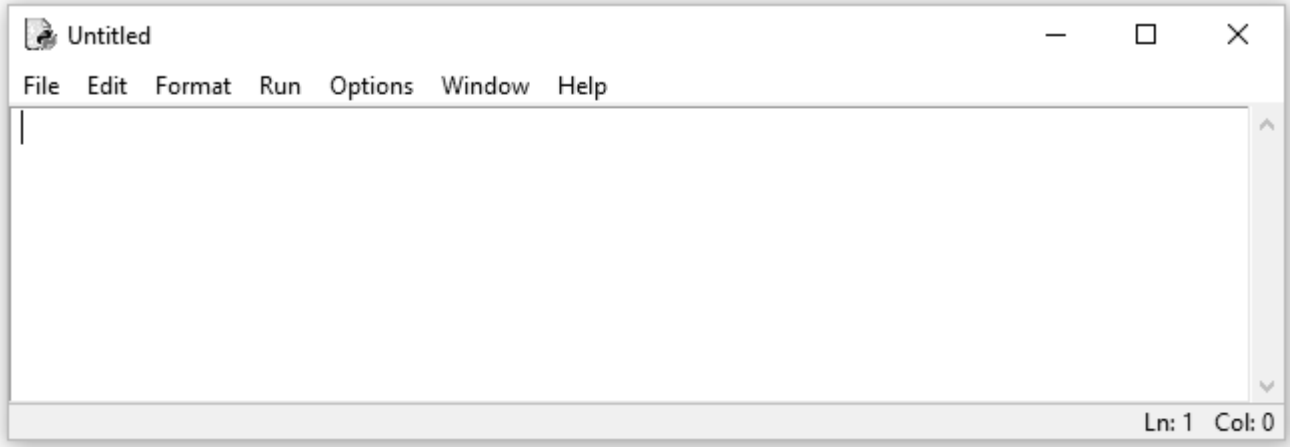
> NOTA: Como ya nos habremos dado cuenta, el inglés es el "idioma oficial" de Python. (De hecho, es el idioma universal para la programación en general, para la informática, y para la ingeniería). Los nombres de las variables, las estructuras de programación, e incluso los comentarios, se escriben en inglés. En el mundo profesional, nunca te encontrarás un programa escrito en español, alemán, o italiano, por lo que vale la pena que te vayas acostumbrando a manejar el inglés al programar.

2.5. NUESTRO PRIMER PROGRAMA EN PYTHON.

Aunque el shell interactivo es muy útil para ejecutar instrucciones de Python una a una, para escribir programas utilizaremos el editor de archivos de Python. El **editor de archivos** es parecido a ciertos editores de texto como el Notepad o el TextMate, pero incluye algunas características específicas que facilitan la escritura de código fuente. Para abrir el editor de archivos en el IDLE, selecciona "File" → "New File".

La ventana que aparece debería contener un cursor que aguarda a que introduzcamos instrucciones. Pero esta ventana es algo distinta a la del shell interactivo, el cual ejecuta una instrucción en cuanto presionas la tecla ENTER. El editor de archivos te permite escribir muchas instrucciones, guardar el archivo, y ejecutar el programa completo. Es fácil diferenciar entre ambos porque la ventana del shell interactivo siempre será aquella con el prompt >>>. El editor de archivos no tiene prompt >>> (ver figura).





Ahora vamos a escribir nuestro primer programa. En el *editor de archivo*, escribe el siguiente código:

```
❶ # This program says hello and asks for my name.

❷ print('Hello world!')
  print('What is your name?')    #asks for their name
❸ myName = input()
❹ print('It is nice to meet you, ' + myName)
❺ print('The length of your name is:')
  print(len(myName))
❻ print('What is your age?')     #asks for their age
  myAge = input()
  print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

Después de haberlo escrito, guárdalo para no tener que volver a escribirlo cada vez que arranques el IDLE. Para ello, en el menú en la parte superior del editor de archivos, selecciona "File" → "Save As". En la ventana emergente introduce `hello.py` como nombre de archivo, y guárdalo en tu carpeta de trabajo.

Es muy recomendable guardar los programas de vez en cuando conforme los escribimos. De esta forma, si el ordenador se bloquea o se apaga, o si accidentalmente salimos del IDLE, no perderemos el código que ya hayamos escrito y guardado. Una vez guardado por primera vez, puedes seleccionar "File" → "Save" (o usar la combinación de teclas CRTL + S) para guardar una nueva versión de tu archivo.

Una vez guardado, vamos a ejecutar nuestro programa. Para ello, selecciona "Run" → "Run Module" (o simplemente presiona la tecla F5). El programa debería ejecutarse en la ventana del shell interactivo que apareció nada más arrancar el IDLE. (Recuerda: los programas se ejecutan desde la ventana del editor, y no desde la ventana del shell interactivo). Escribe tu nombre cuando el programa te lo pida. La salida del programa (tal y como aparece en el shell interactivo) debería ser similar a ésta:

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
======== RESTART: C:\Users\Alejandro\Dropbox\PYTHON\askNameAndAge.py ======
Hello world!
What is your name?
Alex
It is nice to meet you, Alex
The length of your name is:
4
What is your age?
15
You will be 16 in a year.
>>>
```

Cuando no hay más líneas de código que ejecutar, el programa de Python **finaliza**, esto es, deja de ejecutarse. (También suele decirse que el programa **sale**).

Puedes salir del editor de archivos pinchando en la `X` de la parte superior de la ventana. Para volver a cargar un programa ya guardado, selecciona "File" → "Open" en el menú. En la ventana emergente, acude a tu carpeta de trabajo, elige el archivo `hello.py`, y clica en el botón "Open". El programa `hello.py` que habíamos guardado antes debería abrirse en la ventana del editor de archivos.

2.6. DISECCIONAMOS NUESTRO PRIMER PROGRAMA.

Con nuestro programa abierto en el editor de archivos, vamos a echar un vistazo a las instrucciones de Python que hemos usado, analizando lo que hace cada una de las secciones y líneas del código.

COMENTARIOS.

La línea (1) del programa es un **comentario**. En Python, todo texto hasta el final de una línea precedida de una almohadilla (`#`) forma parte de un comentario. Python ignora los comentarios, por lo que podemos usarlos para escribir notas o recordatorios de lo que hace una cierta línea de código.

En ocasiones, los programadores ponen una `#` delante de una línea de código para quitarla temporalmente cuando prueban un programa. Esta práctica es muy útil cuando estamos intentando averiguar por qué no funciona un programa. Después, podemos quitar la `#` cuando queramos restituir la línea que retiramos.

Python también ignora la línea en blanco tras un comentario. Esto permite hacer el código más fácil de leer, como los párrafos de un libro.

LA FUNCIÓN PRINT().

La función `print()` muestra por pantalla *cualquier* valor (entero, flotante, o cadena) que le hayamos introducido dentro del paréntesis. En la sección (2) del programa, la instrucción `print('Hello world!')` significa "Imprime por pantalla el texto de la cadena `'Hello world!'`". Cuando ejecuta esta instrucción, se dice que Python está **llamando** a la función `print()`, y que a dicha función se le está **pasando** la cadena `'Hello world!'` como **argumento**. (En programación, al valor que se le pasa a una función se le denomina *argumento*). Notar que las comillas no se imprimen por pantalla (las comillas solo indican dónde empieza y dónde termina la cadena, pero no forman parte de la cadena).

NOTA: También puedes usar la función `print()` para poner una línea en blanco por pantalla. Para ello, basta con llamar a la función `print()` sin nada entre los paréntesis.

Al escribir un nombre de función, los paréntesis al final permiten identificarlo como el nombre de una función. Esta es la razón por la que se escribe `print()` y no `print`. Más adelante estudiaremos las funciones con mayor detalle.

LA FUNCIÓN INPUT().

La función `input()` espera a que el usuario escriba un texto *con el teclado* y presione la tecla ENTER. En la línea (3) de nuestro programa, la llamada a esta función se evalúa a una *cadena* igual al texto escrito por el usuario, y asigna ese valor a la variable `myName`.

IMPRIMIR EL NOMBRE DEL USUARIO.

En la línea (4), la llamada a `print()` contiene la expresión `'It is good to meet you, ' + myName` dentro de los paréntesis. Recuerda que las expresiones siempre pueden evaluarse a un valor único. Si `'Al'` es el valor almacenado en `myName`, esta expresión se evalúa a `'It is good to meet you, Al'`. A continuación, el programa pasa este valor a la función `print()`, que la imprime por pantalla.

LA FUNCIÓN LEN().

A la función `len()` se le pasa una valor de tipo cadena (o una variable que contenga una cadena), y la función se evalúa a un valor entero igual a número de caracteres de la cadena. Escribe las siguientes expresiones en el shell interactivo para probar esta función:

```
>>> len('hello')
5
>>> len('I am pleased to welcome you in this marvelous morning')
53
>>> len('')
0
```

Como en los ejemplos, la expresión `len(myName)` en la sección (5) de nuestro programa se evalúa a un entero. Después, el programa le pasa este valor a `print()` para mostrarlo por pantalla. Notar que la función `print()` permite que le pasemos tanto enteros como cadenas. Pero observa el error que se produce cuando escribes la siguiente instrucción en el shell interactivo:

```
>>> print('I am '+18+' years old')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print('I am '+18+' years old')
TypeError: can only concatenate str (not "int") to str
```

No es la función `print()` la que produce el error, sino la expresión que le hemos intentando pasar. (Obtendríamos el mismo error si escribiésemos esa expresión sola en el shell interactivo). Python produce este error porque el operador `+` puede usarse para sumar dos números o concatenar dos cadenas, pero no para sumar un entero y una cadena. Podemos arreglar este error usando la versión tipo cadena de `18` en lugar de la versión tipo entero, como explicamos en la siguiente sección.

LAS FUNCIONES STR(), INT(), Y FLOAT().

Si queremos concatenar un entero (por ejemplo, `18`) con una cadena para después pasarle todo este conjunto a la función `print()`, necesitamos obtener el valor `'18'`, que es la forma tipo cadena de `18`.

Para ello usamos la función `str()`, a la que le podemos pasar un entero para que lo evalúe a su versión tipo cadena:

```
>>> str(18)
'18'
>>> print('I am '+str(18)+' years old')
I am 18 years old
```

Las funciones `str()`, `int()`, y `float()` se evaluarán a las formas tipo cadena, entero, y flotante del valor que les pasemos como argumento, respectivamente. Prueba a convertir algunos valores con estas funciones en el shell interactivo, y observa lo que ocurre:

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

La función `str()` es útil cuando tenemos un entero o un flotante que queremos concatenar a una cadena. La función `int()` también es útil si tenemos un número en forma de cadena y lo queremos usar para hacer algunas operaciones matemáticas. Por ejemplo, la función `input()` siempre devuelve una cadena, incluso cuando el usuario inserta un número. Escribe `spam = input()` en el shell interactivo, e introduce `101` cuando quede a la espera de tu respuesta:

```
>>> spam=input()
101
>>> spam
'101'
```

El valor almacenado en la variable `spam` no es el entero `101`, sino la cadena `'101'`. Si queremos realizar operaciones matemáticas usando el valor de `spam`, hemos de usar la función `int()` para obtener la forma entera de `spam`, y después volver a guardar este nuevo valor en la propia variable `spam`:

```
>>> spam=int(spam)
>>> spam
101
```

Ahora ya deberíamos poder usar la variable spam como un entero, y no como una cadena:

```
>>> spam*10/5
202.0
```

Notar que si a `int()` le pasamos un valor que no puede evaluar a un entero, Python mostrará un mensaje de error:

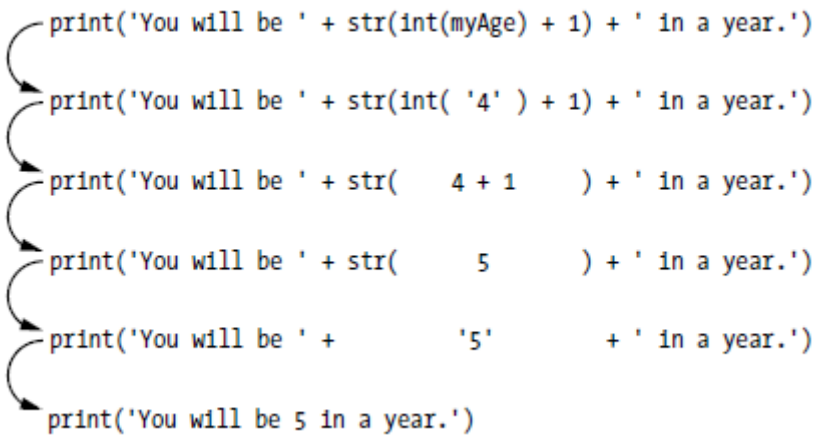
```
>>> int('99.99')
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int('five')
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    int('five')
ValueError: invalid literal for int() with base 10: 'five'
```

La función `int()` también es útil para redondear un número de tipo flotante al entero inmediatamente inferior:

```
>>> int(7.89)
7
>>> int(7.89)+1
8
```

En la sección (6) de nuestro programa usamos las funciones `int()` y `str()` para obtener un valor con el tipo de datos apropiado. La variable `myAge` contiene el valor devuelto por la función `input()`. Como esta función siempre devuelve una cadena, incluso cuando el usuario inserta un número, podemos usar el código `int(myAge)` para devolver un valor entero a partir de la cadena `myAge`. A continuación, el programa le suma 1 a este valor entero, mediante la expresión `int(myAge)+1`. El resultado de esta expresión se le pasa a la función `str()`, mediante la expresión `str(int(myAge)+1)`, para obtener un valor de tipo cadena. Por último, ese valor tipo cadena se concatena con las cadenas `'You will be '` y `' in a year.'` para crear el mensaje final e imprimirlo por pantalla mediante la función `print()`.

Por ejemplo, digamos que como respuesta al `input()`, el usuario proporciona un `4`. En ese caso, la variable `myAge` almacenará el valor `'4'`. Los pasos según los cuales se evaluará la expresión en la tercera línea de la sección (6) del programa serán los que muestra la figura:



Equivalencias de números y textos.

Aunque el valor tipo cadena de un número se considera un valor completamente distinto de su versión entera o punto flotante, un entero puede ser igual a un punto flotante.

Python hace esta distinción porque las cadenas son textos, mientras que los enteros y los flotantes son ambos números.

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

2.7. IMPORTAR MÓDULOS.

Todos los programas de Python pueden llamar a un conjunto básico de funciones, llamadas **funciones integradas** (*built-in* functions), entre las que están las funciones `print()`, `input()`, y `len()` que ya hemos estudiado en secciones previas. Python también viene con un conjunto de módulos llamado **biblioteca estándar**. Cada **módulo** es un programa de Python que contiene un grupo de funciones relacionadas que podemos incorporar en nuestros programas. Por ejemplo, el módulo `math` contiene funciones matemáticas relacionadas (no confundir las funciones matemáticas con los operadores matemáticos que vimos en la sección 2.1), el módulo `random` incluye funciones relacionadas con números aleatorios, etc. Pero antes de poder usar las funciones de un módulo en nuestro programa, debemos importar el módulo mediante una sentencia `import`. En Python, una sentencia `import` consiste en:

- La palabra reservada `import`.
- El nombre del módulo.
- Opcionalmente, más nombres de módulos, separados mediante comas.

Después de haber importado un módulo, ya podemos usar las funciones que incluye ese módulo.

Vamos a probar el módulo `math`, el cual nos da acceso a la función `math.sqrt()` que permite obtener la raíz cuadrada de un número. Para ver esta función en acción, vamos a crear un programa que obtenga las raíces de una **ecuación cuadrática** de la forma $ax^2 + bx + c = 0$. Como sabemos, las soluciones vienen dadas por la **fórmula cuadrática**:

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

Escribe el siguiente código en el editor de archivos de Python y guárdalo como `cuadraticFormula.py`.

```
# This program uses the cuadratic formula to find
# the real roots of the cuadratic equation ax^2+bx+c=0.

import math

#asks user to insert the coefficients a, b, and c.
print ('This program fins the roots of ax^2+bx+c=0')
print('Please, insert coefficient a:')
a = float(input())
print('Please, insert coefficient b:')
b = float(input())
print('Please, insert coefficient c:')
c = float(input())

# Finds the real roots using the cuadratic formula.
d = (b*b)-4*a*c  #Finds the determinant.
if d < 0:
    print('This equation has no real roots')
else:
    x1 = (-b+math.sqrt(d))/2*a
    print ('First real root: x1 = ' + str(x1))
    x2 = (-b-math.sqrt(d))/2*a
    print ('Second real root: x2 = ' + str(x2))
```

Si usamos este programa para resolver la ecuación $x^2 - 5x + 4 = 0$, la salida debería ser la siguiente:

```
This program finds the roots of ax^2+bx+c=0
Please, insert coefficient a:
1
Please, insert coefficient b:
-5
Please, insert coefficient c:
4
First real root: x1 = 4.0
Second real root: x2 = 1.0
```

Por supuesto, el módulo `math` incorpora muchas más funciones matemáticas, aparte de la función `math.sqrt()`. Para ver alguna de las funciones que contiene, escribe lo siguiente en el shell interactivo:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/2)
1.0
>>> math.cos(math.pi/4)
0.7071067811865476
>>> math.e
2.718281828459045
>>> math.exp(1)
2.718281828459045
>>> math.exp(3)
20.085536923187668
>>> math.log(math.e)
1.0
>>> math.asin(1)
1.5707963267948966
>>> math.pi/2
1.5707963267948966
```

Para acceder al resto de funciones disponibles en el módulo `math`, notar que al escribir `math.` en el shell interactivo, aparece una ventana contextual que muestra una lista con todas las funciones incluidas en el módulo. Navega por esta lista y observa la cantidad de operaciones matemáticas que el módulo `math` pone a disposición de nuestros programas:

Por último, mostramos un ejemplo de una sentencia `import` que importa cuatro módulos distintos:

```
import random, sys, os, math
```

Después de escribir esta sentencia, podemos usar cualquiera de las funciones de estos cuatro módulos. Aprenderemos más sobre estos módulos más adelante en este texto.

SENTENCIAS FROM IMPORT.

Una forma alternativa de la sentencia `import` se compone de la palabra reservada `from`, seguida del nombre del módulo, la palabra reservada `import`, y un asterisco; por ejemplo, `from math import *`.

Con esta forma de la sentencia `import`, las llamadas a las funciones en el módulo `math` no necesitan el prefijo `math.` delante de ellas. Sin embargo, usar el nombre completo hace el código mucho más legible, por lo que es mejor usar la forma normal de la sentencia `import`.

2.8. USAR UN PROGRAMA COMO UNA CALCULADORA.

Una aplicación muy común de Python es emplearlo como calculadora programable para computar operaciones matemáticas y fórmulas. A menudo, las fórmulas matemáticas incluyen funciones como el seno, coseno, tangente, seno hiperbólico, coseno hiperbólico, logaritmos, exponenciales, raíces cuadradas, etc. En nuestra calculadora de bolsillo tenemos botones especiales para acceder a estas funciones. De manera similar, en Python disponemos de funciones preconstruidas para evaluar este tipo de funciones matemáticas. En principio, podríamos escribir nuestro propio programa para evaluar, digamos, la función $\sin(x)$, pero hacerlo de forma eficiente es un problema bastante complejo. Los programadores expertos han trabajado en este problema durante décadas, y ha implementado sus soluciones en programas que podemos reutilizar.

Por ejemplo, considera la fórmula para la altura y de una pelota en caída libre, con la velocidad inicial v_o apuntando hacia arriba:

$$y = v_o t - gt^2/2$$

, donde $g = 9{,}81\ m/s^2$ es la aceleración de la gravedad y t es el tiempo. ¿Cuánto *tiempo* tarda la pelota en alcanzar una cierta altura h? La respuesta es fácil de hallar. Cuando $y = h$ tenemos que:

$$h = v_o t - \frac{1}{2}gt^2 \qquad \Rightarrow \qquad \frac{1}{2}gt^2 - v_o t + h = 0$$

En esta fórmula reconocemos la ecuación cuadrática $ax^2 + bx + c = 0$ para la incógnita t. Usando la fórmula cuadrática $x = \left(-b \pm \sqrt{b^2 - 4ac}\right)/2a$, encontramos que:

$$t_1 = \left(v_o - \sqrt{v_0^2 - 2gh}\right)/g \qquad t_2 = \left(v_o + \sqrt{v_0^2 - 2gh}\right)/g$$

(Tenemos dos soluciones porque la pelota alcanza la altura h al subir, y posteriormente, al volver a caer).

Para evaluar estas expresiones para t_1 y t_2 mediante un programa, necesitamos tener acceso a la función raíz cuadrada, que como sabemos, está contenida en el módulo `math`. Escribe el siguiente programa en el editor de archivos, y guárdalo como `thrownBall.py`.

```
#Data from user:
v0 = 5  #Initial velocity in m/s.
h = 0.2 #Height in meters.

import math

g = 9.81 #Gravity acceleration in m/s^2.
t1 = (v0 - math.sqrt(v0**2 - 2*g*h))/g
t2 = (v0 + math.sqrt(v0**2 - 2*g*h))/g
print ('At t = '+str(t1)+' s and t = '+str(t2)+ ' s, the height is '
       +str(h)+' m.')
```

Notar que en este caso, el programa no le pide al usuario los datos de velocidad inicial y altura alcanzada, sino que es el usuario el que inserta dichos datos escribiéndolos directamente dentro del programa. La salida de este programa debería ser:

```
At t = 0.041706372498337634s and t = 0.9776616193467182s, the height is 0.2m.
```

Como segundo ejemplo, vamos a escribir un programa que convierta una temperatura de grados Celsius (o centígrados) a grados Fahrenheit y a Kelvin.

La escala Celsius es una escala de temperaturas que asigna a las temperaturas de congelación y de ebullición del agua unos valores arbitrarios de $0\,°C$ y $100\,°C$, respectivamente. La escala Kelvin es una escala de temperaturas absoluta referida a la temperatura a la que cesa todo el movimiento microscópico de las moléculas de un gas. A esta temperatura especial se la denomina cero absoluto, y define el punto cero de la escala kelvin, esto es $0\,K$, lo que equivale aproximadamente a $-273\,°C$. Un kelvin tiene el mismo tamaño que un grado Celsius, pero las temperaturas kelvin se miden hacia arriba desde el cero absoluto, en vez de medirse hacia arriba y hacia abajo desde el punto de congelación del agua. Por último, la escala de temperatura Fahrenheit se define de forma que el agua se congela a $32\,°F$ y hierve a $212°F$. Las fórmulas de conversión son:

$$T(K) = T(°C) + 273$$

$$T(°F) = \frac{9}{5}T(°C) + 32$$

El programa sería similar al siguiente:

```
C = 25  #Celsius temperature typed by user

K = C + 273
F = (9/5)*C + 32
print('A temperature of '+str(C)+' °C equals to '+str(K)+' kelvin and '
      +str(F)+ ' °F.')
```

La salida del programa (para $25\,°C$) debería ser:

```
A temperature of 25 °C equals to 298 kelvin and 77.0 °F.
```

En física y química, las ecuaciones solo son válidas si la temperatura está expresada en kelvin. Por el contrario, en el lenguaje cotidiano las temperaturas se expresan en grados Celsius (Europa) o en grados Fahrenheit (Estados Unidos).

2.9. EJERCICIOS DEL CAPÍTULO 2.

Ejercicio 2.1. Escribe un programa que le pida al usuario que introduzca por el teclado dos números, que sume estos números, y que muestre el resultado por pantalla. Guarda el programa como `Ejer2.1.py`.

Ejercicio 2.2. Escribe un programa que te pida tu nombre y el del compañero que tengas más cerca. Este programa debe calcular la longitud de ambos nombres, concatenar ambos nombres, y calcular la longitud de esta cadena combinada. El programa debe mostrar por pantalla todos estos resultados. Guarda el programa como `Ejer2.2.py`.

Ejercicio 2.3. Escribe un programa que te pida el año actual, y calcule tu edad actual en base al año en el que naciste. Ten en cuenta que el programa solo calcula una aproximación a tu edad, porque no estamos considerando el día ni el mes actual, ni el día ni el mes en el que naciste. Guarda el programa como `Ejer2.3.py`.

Ejercicio 2.4. Imagina que trabajas en la ventanilla de un banco. Vas a recibir a tres clientes, que harán tres ingresos. Para hacer cada ingreso, el programa pregunta cuánto dinero quiere ingresar el cliente, y el cliente responde a través del teclado. En el momento que el usuario responde, se considera que el ingreso se ha realizado. El objetivo es que le programa haga la suma de todas las cantidades ingresadas y la muestre por pantalla, indicando un mensaje similar a éste:

```
Hoy se han ingresado 1500 € en total.
```

Guarda el programa como `Ejer2.4.py`.

Ejercicio 2.5. Vamos a hacer una modificación del programa 2.5. Supón que además de realizar tres ingresos, llegan dos clientes que realizan dos retiradas de efectivo. (Para retirar efectivo, el programa pregunta cuánto dinero quiere retirar el cliente, y el cliente responde mediante el teclado. Una vez ha respondido, se considera que la retirada de efectivo se ha realizado). Asume que primero se hacen los tres ingresos, y luego las dos retiradas de efectivo. La tarea del programa es hacer el balance diario del banco, esto es $(ingresos - retiradas)$, y mostrar el resultado por pantalla. Utiliza un mensaje similar a éste:

```
Hoy se han ingresado 1500 € en total, y se han retirado 950 € en total. El
balance final es de 550 €.
```

(Notar que el balance final puede ser una cantidad negativa, si las retiradas exceden a los ingresos). Guarda el programa como `Ejer2.5.py`.

Ejercicio 2.6. Escribe un programa en el que insertes las notas de los tres exámenes que has hecho de matemáticas durante el trimestre, y que te calcule la nota media de los exámenes. El primer examen cuenta el 20% de la nota media, el segundo el 30%, y el último el 50%. Cuando tengas la nota media ponderada de los tres exámenes, inserta el número de negativos que tienes por no haber traído los deberes. Cada negativo resta 0,1 puntos a la nota media de los exámenes. Al final, el programa muestra por pantalla la nota final trimestral. Guarda el programa como `Ejer2.6.py`.

Ejercicio 2.7. Usando el Python como calculadora, calcula la tasa de interés anual. Si el depósito ingresado en el banco es A, transcurridos n años esta cantidad de dinero se habrá convertido en:

$$A\left(1+\frac{p}{100}\right)^n$$

, donde p es el tipo de interés ofrecido por el banco. Haz un programa que compute cuánto crecerá un depósito de 1000 € si el interés está al 5%. Guarda el programa como `Ejer2.7.py`.

Ejercicio 2.8.
(a) Dejamos caer una pelota desde una torre de altura h. Su velocidad inicial es cero, y al ser liberada, la pelota se acelera hacia abajo debido a la gravedad a razón de $g = 9{,}81\ m/s^2$. El objetivo es escribir un programa que le pida al usuario la altura h de la torre en metros y un intervalo temporal t en segundos, y que imprima por pantalla la altura a la que se encuentra la pelota en relación al nivel del suelo un tiempo t después de haber sido soltada. Ignorando la resistencia del aire, la respuesta viene dada por la fórmula $y(t) = -gt^2/2 + h$ (válida para un sistema de referencia con origen en el suelo, donde $y = 0$, tal y como nos pide el ejercicio). Guarda el programa como `Ejer2.8a.py`.
(b) Dejamos caer de nuevo una pelota desde una torre de altura h con una velocidad inicial cero. Escribe un programa que le pida al usuario la altura en metros de la torre, y que calcule e imprima el tiempo que tarda la pelota en llegar al suelo ($y = 0$), ignorando la resistencia del aire. Usa tu programa para calcular el tiempo que tardará una pelota liberada desde una altura de $100\ m$ en llegar al suelo. Guarda el programa como `Ejer2.8b.py`.

Ejercicio 2.9. De física sabrás que el movimiento de proyectil es una combinación de un movimiento a velocidad constante en la dirección horizontal, y de un movimiento de caída libre (esto es, un movimiento bajo la única influencia de la gravedad) en la dirección vertical. (En este caso, estamos ignorando la resistencia del aire). Cuando se combinan los dos movimientos, la trayectoria resultante es una parábola. Imagina que un cañón dispara un obús con una cierta velocidad inicial $\vec{v}_0$ y con un cierto ángulo θ desde la horizontal. La componente horizontal de la velocidad inicial es $v_{0x} = v_0 \cos\theta$, y la componente vertical es $v_{0y} = v_0 \sin\theta$. Situando el origen de nuestro sistema de coordenadas en el punto en el que el proyectil comienza su vuelo, el valor de la coordenada x del proyectil en cualquier instante de tiempo posterior es $x(t) = v_{0x}t$, y el de la coordenada y es $y(t) = v_{0y}t - \frac{1}{2}gt^2$, donde $g = 9{,}8\ m/s^2$ es la aceleración gravitacional. Usando estas ecuaciones, podemos calcular el tiempo total de vuelo t_{total}, la altura máxima h alcanzada, y el alcance horizontal d del proyectil:

$$t_{total} = \frac{2v_0 \sin\theta}{g} \qquad h = \frac{(v_0 \sin\theta)^2}{2g} \qquad d = \frac{v_0^2 \sin 2\theta}{g}$$

Escribe un programa que, dados los valores de la rapidez inicial v_0 (en metros por segundo) y el ángulo θ (en grados) con el que se dispara el proyectil, calcule el tiempo total de vuelo (en segundos), la altura máxima alcanzada (en metros), y el alcance máximo horizontal (en metros). Guarda el programa como `Ejer2.9.py`.

NOTA: Las funciones trigonométricas seno y coseno están contenidas el módulo `math`, y se escriben como `math.sin()` y `math.cos()`. Estas funciones necesitan como argumento el ángulo en *radianes*. Aquí tienes un ejemplo de utilización de estas funciones. (La función `math.pi` (sin argumentos) simplemente proporciona el número π).

```
#This program computes the cosine and the sine of a given angle.

import math
#To use trigonometric functions, we need to import math module.

print('Please insert angle (in degrees):')
angle = input()

#convert degrees to radians (math.cos() and math.sin() require radians):
angle = (float(angle)*math.pi)/180

print('The cosine of the inserted angle is:')
cosine = math.cos(angle)
print(cosine)
print('The sine of the inserted angle is:')
sine = math.sin(angle)
print(sine)
```

Ejercicio 2.10. Un satélite debe ser lanzado a una órbita circular alrededor de la Tierra, de forma que orbite al planeta una vez cada T segundos. Como se puede demostrar, la altitud h sobre la superficie de la Tierra que debe tener el satélite debe ser:

$$h = \left(\frac{GMT^2}{4\pi^2}\right)^{1/3} - R$$

, donde $G = 6{,}67 \times 10^{-11}\ m^3 kg^{-1} s^{-2}$ es la constante gravitacional de Newton, $M = 5{,}97 \times 10^{24}\ kg$ es la masa de la Tierra, y $R = 6371\ km$ su radio. Escribe un programa que reciba el valor de T deseada para la órbita, y que calcule e imprima la altitud correcta para la órbita en metros. Usa tu programa para calcular la altitud a la que deben lanzarse los satélites que orbiten la Tierra una vez al día (las llamadas órbitas geosíncronas), una vez cada 90 minutos, una vez cada 60 minutos, y una vez cada 45 minutos. ¿Qué concluyes de este último resultado? Guarda el programa como `Ejer2.10.py`.

Ejercicio 2.11. La función Gaussiana:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-m}{s}\right)^2}$$

, es una de las más usadas en ciencia. Los parámetros m y s (ambos mayores que cero) son dos números reales. Usando las distintas funciones incorporadas en el módulo `math`, haz un programa que evalúe la función cuando $m = 0,\ s = 2$, y $x = 1$. Guarda el programa como `Ejer2.11.py`.

Ejercicio 2.12. Órbitas planetarias.
En general, la órbita que sigue un cuerpo en interacción gravitacional con otro (como en el caso de un planeta y el Sol) es elíptica, de forma que el planeta a veces está más cerca y a veces está más lejos del Sol. Si se nos da la distancia l_1 de máxima aproximación del planeta al sol (el perihelio) y su velocidad lineal v_1 en el perihelio, podemos calcular cualquier propiedad de la órbita a partir de ellas.

(a) La segunda ley de Kepler dice que la distancia l_2 y la velocidad lineal v_2 en el punto más alejado (el afelio) satisfacen $l_2 v_2 = l_1 v_1$. Al mismo tiempo, la energía total (cinética más potencial) de un planeta con una velocidad v a una distancia r del Sol es:

$$E = \frac{1}{2}mv^2 - G\frac{mM}{r}$$

, donde m es la masa del planeta, $M = 1{,}9891 \times 10^{30}\ kg$ es la masa del Sol, y $G = 6{,}6738 \times 10^{-11}\ m^3 kg^{-1} s^2$ es la constante de gravitación de Newton. Aplicando la conservación de la energía, se puede demostrar que v_2 viene dada por la raíz menor de la ecuación cuadrática:

$$v_2^2 - G\frac{mM}{l_1 v_1} v_2 - \left[v_1^2 - \frac{2GM}{l_1}\right] = 0$$

Una vez tenemos v_2, podemos calcular l_2 usando $l_2 v_2 = l_1 v_1$.

(b) Dados los valores de $v_1,\ l_1$, y l_2 podemos obtener otros parámetros de la órbita, que derivan de las leyes de Kepler y de las propiedades de las elipses:
Semieje mayor:

$$a = \frac{1}{2}(l_1 + l_2)$$

Semieje menor:

$$b = \sqrt{l_1 l_2}$$

Periodo orbital:

$$T = \frac{2\pi ab}{l_1 v_1}$$

Excentricidad orbital:

$$e = \frac{l_2 - l_1}{l_2 + l_1}$$

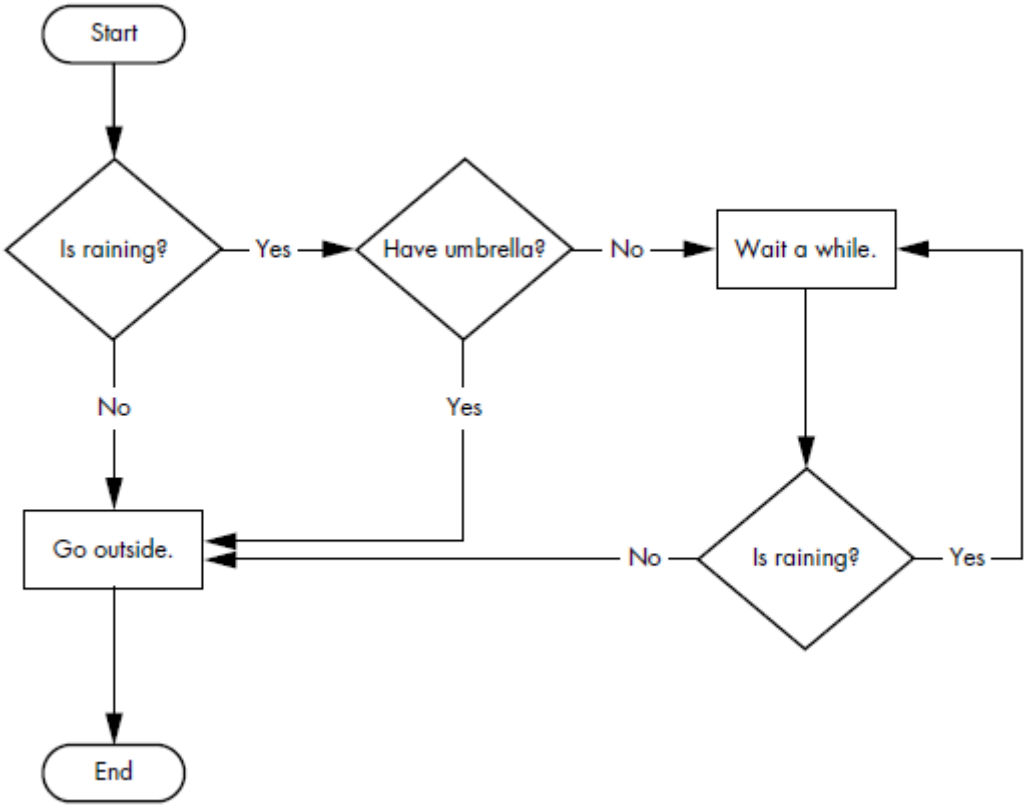
Escribe un programa que le pida al usuario la distancia al Sol y la velocidad en el perihelio, y que calcule e imprima las cantidades $v_2,\ l_2,\ T$, y e.

(c) Comprueba tu programa calculando las propiedades de las órbitas de la Tierra ($l_1 = 1{,}4710 \times 10^{11}\ m$ y $v_1 = 3{,}0287 \times 10^4\ m\ s^{-1}$) y del cometa Halley ($l_2 = 8{,}7830 \times 10^{10}\ m$ y $v_1 = 5{,}4529 \times 10^4\ m\ s^{-1}$). Busca en libros o Internet el resto de datos que necesites. Deberías hallar que el periodo orbital de la Tierra es, por supuesto, 1 año, y que el del cometa Halley es de aproximadamente 76 años.

3. CONTROL DE FLUJO.

Ahora ya conocemos algunas de las instrucciones más básicas de Python, y también sabemos que un programa no es más que una serie de instrucciones. Los programas se hemos escrito hasta ahora son secuenciales: el flujo de ejecución comienza en la primera instrucción, y continúa hacia abajo hasta llegar a la última instrucción, tras la cual termina el programa. Sin embargo, la verdadera programación no consiste en ejecutar una instrucción tras otra. En vez de eso, y basándose en la forma en la que se evalúan unas ciertas expresiones, un programa puede decidir saltarse algunas instrucciones, repetirlas, o elegir que se ejecute una de entre varias instrucciones. De hecho, casi nunca querremos que un programa ejecute de principio a fin cada una de las líneas de su código. Las **sentencias de control de flujo** nos permiten decidir qué instrucciones se ejecutarán bajo qué condiciones.

Estas sentencias de control de flujo se representan mediante símbolos en lo que se denomina un **diagrama de flujo**. Por ejemplo, la figura muestra el diagrama de flujo que nos permite decidir qué hacer si está lloviendo. Sigue el camino indicado por las flechas comenzando en el símbolo "Start" y terminando en el símbolo "End".



En un diagrama de flujo suele haber más de un camino posible para ir desde el principio hasta el final. Lo mismo sucede con las líneas de código en un programa. Los diagramas de flujo representan estos puntos de bifurcación con rombos, mientras que otros pasos se representan con rectángulos. Los puntos de inicio y final se representan mediante rectángulos redondeados.

Pero antes de aprender las sentencias de control de flujo, debemos saber cómo representar estas opciones de *si* o *no*, y cómo escribir estos puntos de bifurcación mediante código Python.

3.1. VALORES BOOLEANOS.

Mientras que los tipos de datos entero, punto – flotante, y cadena pueden tener un número infinito de valores posibles, el tipo de datos **Booleano** solo puede tomar dos valores: `True` (verdadero) y `False`

(falso).[2] Cuando los escribimos en un programa de Python, los valores `True` y `False` no llevan las comillas que solemos poner delante y detrás de las cadenas, y además, siempre se escriben con T y F mayúsculas (y el resto de la palabra en minúsculas). Inserta las siguientes expresiones en el shell interactivo (algunas de las instrucciones son intencionadamente incorrectas, y producirán un mensaje de error):

```
❶ >>> spam = True
  >>> spam
  True
❷ >>> true
  Traceback (most recent call last):
    File "<pyshell#2>", line 1, in <module>
      true
  NameError: name 'true' is not defined
❸ >>> True = 2+2
  SyntaxError: can't assign to keyword
```

Como cualquier otro valor, los valores Booleanos pueden usarse en expresiones y almacenarse en variables, como en (1). Si no escribimos el valor con la primera letra en mayúscula (2), o si intentamos usar `True` y `False` como nombres de variables (3), Python nos dará un mensaje de error.

3.2. OPERADORES DE COMPARACIÓN.

Los **operadores de comparación** nos permiten comparar dos valores y evaluar una expresión de comparación a un valor Booleano único. La tabla a continuación lista todos los operadores de comparación:

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Estos operadores evalúan una expresión a verdadero (`True`) o falso (`False`) dependiendo de los valores que les demos. Vamos a probar algunos de ellos, comenzando con `==` y `!=`.

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

Como cabía esperar, `==` (igual a) se evalúa a `True` cuando los valores a ambos lados son iguales, y `!=` (no es igual a) se evalúa a `True` cuando los dos valores son distintos. Los operadores `==` y `!=` pueden usarse con valores de cualquier tipo de datos.

[2] Booleano se escribe con mayúscula porque el nombre de este tipo de datos proviene del nombre del matemático británico George Boole, inventor del álgebra que permite tratar con valores booleanos.

Escribe las siguientes expresiones en el shell interactivo, y observa cómo se evalúan. Notar que un valor entero o de tipo punto – flotante siempre será diferente a un valor de tipo cadena. Por ejemplo, la expresión `42 == '42'` se evalúa a `False` porque Python considera que el entero `42` es diferente de la cadena `'42'`.

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
>>> 42 == '42'
False
```

Por su parte, los operadores <, >, <=, y >= solo funcionan correctamente con valores enteros y flotantes (sin embargo, ver último párrafo de esta sección):

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
❶ >>> eggCount <= 42
True
>>> myAge = 29
❷ >>> myAge >= 10
True
```

A menudo usaremos operadores de comparación para comparar el valor de una variable con algún otro valor, como en las expresiones `eggCount <= 42` y `myAge >= 10` en (1) y (2), respectivamente. Veremos muchos más ejemplos más adelante, cuando estudiemos las sentencias de control de flujo.

La diferencia entre los operadores = y ==.

Puede que hayas notado que el operador == (igual a) tiene dos signos de igualdad, mientras que el operador = solo tiene un signo de igualdad. Es fácil confundir estos dos operadores, así que asegúrate de recordar estas indicaciones:

- El operador == (igual a) sirve para preguntar si dos valores son iguales entre sí.
- El operador = (asignación) sirve para adjudicar el valor escrito a su derecha a la variable situada a su izquierda.

A modo de ejemplo, introduce el siguiente código en el shell interactivo:

```
>>> spam = 42
>>> spam == 43
```

La primera instrucción le asigna a la variable spam el valor entero 42, mientras que la segunda comprueba si el valor almacenado en spam es igual al valor entero 43. ¿Cómo se evaluará esta segunda expresión?

Para terminar, vale la pena introducir algunas expresiones de comparación más en el shell interactivo, para ver a qué valor Booleano se evalúan. ¿Hay alguna razón lógica para que se evalúen de esta forma?

```
>>> 'Anthony' > 'Sam'
False
>>> 'abcd' < 'z'
True
>>> -5 < -2
True
>>> abs(-5) < abs(-2)
False
>>> True > False
True
>>> True == 'True'
False
>>> False == false
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    False == false
NameError: name 'false' is not defined
```

3.2. OPERADORES BOOLEANOS.

Los tres **operadores Booleanos** (`and`, `or`, y `not`) se usan para comparar valores Booleanos. Al igual que los operadores de comparación, los operadores Booleanos evalúan las expresiones que los contienen a un valor Booleano único.

OPERADORES BINARIOS (AND Y OR).

Los operadores `and` (y) y `or` (o) siempre reciben dos valores Booleanos (o dos expresiones que se reduzcan a un valor Booleano), razón por las que se les considera **operadores binarios**. El operador `and` evalúa una expresión a `True` si los dos valores Booleanos que opera son `True`; en otro caso, se evalúa a `False`. Vamos a insertar algunas expresiones en el shell interactivo para ver este operador en acción:

```
>>> True and True
True
>>> True and False
False
```

Una **tabla de verdad** muestra cada posible resultado de un operador Booleano. La siguiente tabla muestra la tabla de verdad del operador `and`.

Expression	Evaluates to...
True and True	True
True and False	False
False and True	False
False and False	False

Por su parte, el operador `or` evalúa una expresión a `True` si alguno de los dos valores Booleanos que opera es `True`. Si los dos son `False`, se evalúa a `False`:

```
>>> False or True
True
>>> False or False
False
```

La siguiente tabla de verdad muestra las salidas posibles del operador `or`:

Expression	Evaluates to...
True or True	True
True or False	True
False or True	True
False or False	False

EL OPERADOR NOT.

Al contrario que los operadores `and` y `or`, el operador `not` solo actúa sobre un único valor Booleano (o una única expresión Booleana). El operador `not` simplemente se evalúa al opuesto del valor Booleano proporcionado.

```
>>> not True
False
❶ >>> not not not not True
True
```

Como las dobles negaciones propias del lenguaje oral y escrito, podemos combinar varios operadores `not` en una sola expresión, como en (1). Sin embargo, rara vez necesitaremos hacer esto en nuestros programas.

La siguiente figura muestra la tabla de verdad del operador `not`:

Expression	Evaluates to...
not True	False
not False	True

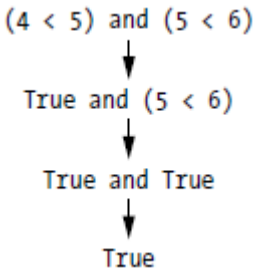
3.3. COMBINAR OPERADORES BOOLEANOS Y DE COMPARACIÓN.

Como los operadores de comparación se evalúan a valores Booleanos, podemos usarlos en expresiones que incluyan operadores Booleanos.

Recordemos que los operadores `and`, `or`, y `not` se denominan operadores Booleanos porque siempre operan sobre los valores Booleanos `True` y `False`. Aunque expresiones como `4 < 5` no son valores Booleanos propiamente dichos, sí que son expresiones que se evalúan a valores Booleanos. Escribe en el shell interactivo las siguientes expresiones Booleanas con operadores de comparación:

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

El ordenador evaluará primero la expresión a la izquierda, y después la expresión a la derecha. Cuando conozca el valor de ambas expresiones, evaluará la expresión completa a un valor Booleano único. A modo de ejemplo, la figura muestra el proceso de evaluación de la expresión `(4 < 5) and (5 < 6)`.



También podemos usar múltiples operadores Booleanos en una expresión, junto con los operadores de comparación:

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

Los operadores Booleanos se rigen por una precedencia, como los operadores matemáticos. Después de que se evalúen todos los operadores matemáticos y de comparación, Python evalúa los operadores `not` en primer lugar, luego los operadores `and`, y finalmente los operadores `or`.

3.4. CONCEPTOS BÁSICOS DE CONTROL DE FLUJO.

Las **sentencias de control de flujo** suelen comenzar con una parte llamada **condición**, y siguen con un **bloque de código** llamado **cláusula**. Antes de estudiar las sentencias de control de flujo, vamos a explicar qué son las condiciones y los bloques de código (cláusulas).

CONDICIONES.

Todas las expresiones Booleanas que hemos visto hasta ahora pueden considerarse condiciones. Las condiciones no son más que expresiones, a las que les damos otro nombre en el contexto de las sentencias de control de flujo. Las condiciones siempre se evalúan a un valor Booleano, `True` o `False`. Una sentencia de control de flujo decide qué hacer basándose en si su condición se evalúa a `True` o a `False`.

BLOQUES DE CÓDIGO (CLÁUSULAS).

Las líneas de código Python pueden agruparse en *bloques*. Podemos saber dónde empieza y termina un bloque mediante la tabulación de sus líneas de código. Hay tres reglas que debemos seguir a la hora de construir bloques:

1) Los bloques comienzan allí donde la tabulación aumenta.
2) Los bloques pueden contener otros bloques.
3) Los bloques terminan allí donde la tabulación disminuye a cero o a la tabulación del bloque que los contiene.

Los bloques son más fáciles de entender mediante ejemplos. Vamos a tratar de identificar los bloques en el siguiente programa:

```
print('Insert your name:')
name = input()
print('Insert your password:')
password = input()

if name == 'Mary':
❶    print('Hello Mary.')
     if password == 'swordfish':
❷        print('Access granted.')
     else:
❸        print('Wrong password.')
else:
❹    print('Hello stranger.')
```

El primer bloque de código (1) comienza en la línea `print('Hello Mary.')` y contiene todas las líneas que le siguen, hasta el segundo `else`. Dentro del bloque (1) hay otros dos bloques: el bloque (2), que solo tiene una línea dentro de él: `print('Access granted.')`, y el bloque (3), que también contiene una sola línea de código: `print('Wrong password.')`. El cuarto bloque ya no pertenece al bloque (1), porque no está tabulado. Se trata de un nuevo bloque que solo contiene una línea de código: `print('Hello stranger.')`.

3.5. EJECUCIÓN DE PROGRAMAS.

En el programa `hello.py` del capítulo previo, Python comenzaba ejecutando las instrucciones al principio de programa y continuaba hacia abajo, ejecutando una instrucción tras otra hasta llegar a la última. La **ejecución del programa** (o simplemente, la **ejecución**) es el término que se utiliza para indicar qué instrucción es la siguiente en ejecutarse. Si imprimimos el código del programa en un papel y ponemos nuestro dedo en cada línea que se va ejecutando, nuestro dedo actuaría como la ejecución del programa.

Pero no todos los programas se ejecutan secuencialmente desde la primera hasta la última línea de código. Si usamos nuestro dedo para rastrear la ejecución de un programa con sentencias de control de flujo, veremos que nuestro dedo debe saltar de una línea a otra dependiendo de cómo se evalúen las condiciones de esas sentencias, y probablemente nos saltaremos algunos bloques o repetiremos la ejecución de otros.

3.6. SENTENCIAS DE CONTROL DE FLUJO.

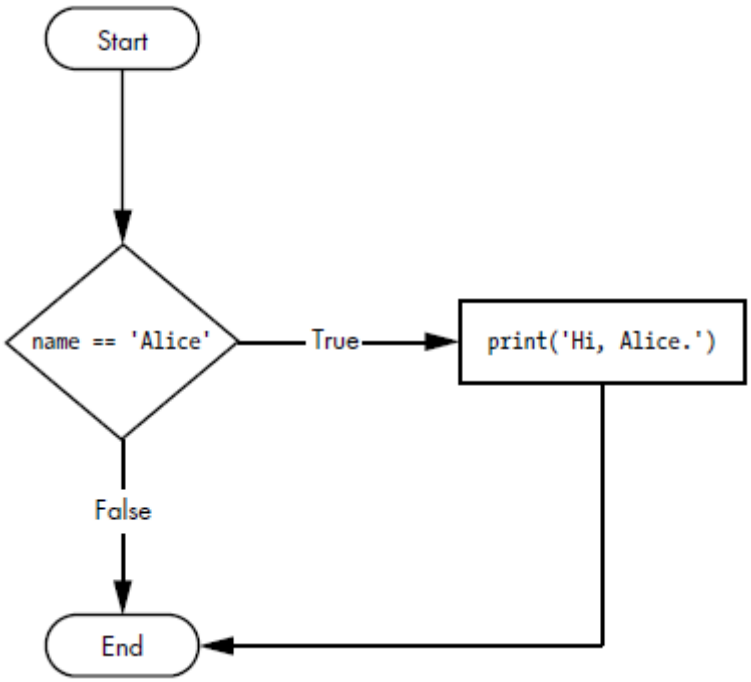
A continuación, vamos a estudiar la parte más importante del control de flujo: las propias sentencias de control de flujo. Estas sentencias representan las formas diagonales que vimos en el diagrama de flujo al principio del capítulo.

SENTENCIAS IF.

Uno de los tipos más comunes de sentencias de control de flujo son las sentencias `if`. La cláusula de una sentencia `if` (es decir, el bloque de código que sigue a la sentencia `if`) se ejecutará si la condición de la sentencia se evalúa a `True`. Por el contrario, la cláusula no se ejecutará si la condición se evalúa a `False`.

En palabras, como el término inglés "if" es el equivalente al "si" condicional en castellano, una sentencia `if` puede leerse como: "Si esta condición es cierta, ejecuta el código de la cláusula". En Python, sentencia `if` consiste en:

- La palabra reservada `if`.
- Una condición (esto es, una expresión que se evalúa a `True` o `False`).
- Dos puntos (`:`).
- Empezando en la siguiente línea, un bloque de código tabulado (al que se denomina cláusula `if`).



Por ejemplo, digamos que comprueba si el nombre de una cierta persona es Alice. (Imagina que previamente el programa ya asignó un valor a la variable `name`).

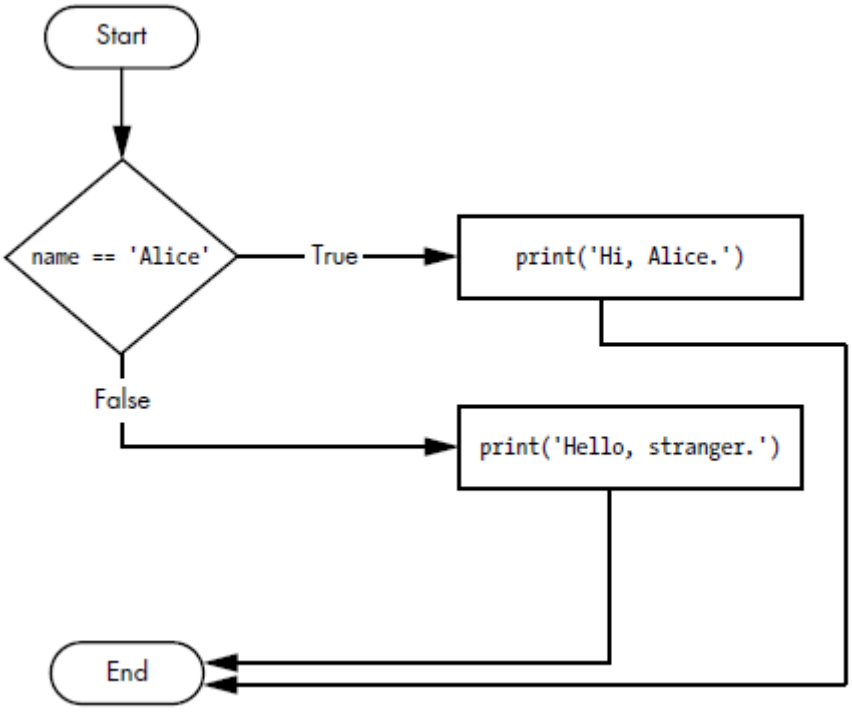
```
if name == 'Alice':
    print('Hi, Alice.')
```

Todas las sentencias de control de flujo terminan con dos puntos, y están seguidas por un nuevo bloque de código (la cláusula). La cláusula de esta sentencia `if` es el bloque de código con la instrucción `print('Hi, Alice.')`. La figura muestra el diagrama de flujo de este código.

SENTENCIAS ELSE.

La cláusula de una sentencia `if` puede venir seguida opcionalmente por una sentencia `else`. La cláusula de la sentencia `else` solo se ejecuta cuando la condición de la sentencia `if` se evalúa a `False`. En palabras, como "else" significa "si no", una sentencia `else` puede leerse como: "Si esta condición es cierta, ejecuta este código. Si no, ejecuta este otro código". Una sentencia `else` no tiene una condición propia (depende de la condición de su sentencia `if` asociada). Escrita en Python, una sentencia `else` siempre consiste en lo siguiente:

- La palabra reservada `else`.
- Dos puntos.
- Empezando en la siguiente línea, un bloque de código tabulado (la cláusula `else`).



Volviendo al ejemplo de Alice, vamos a analizar un código que usa una sentencia `else` para ofrecer un saludo diferente si el nombre del usuario no es Alice:

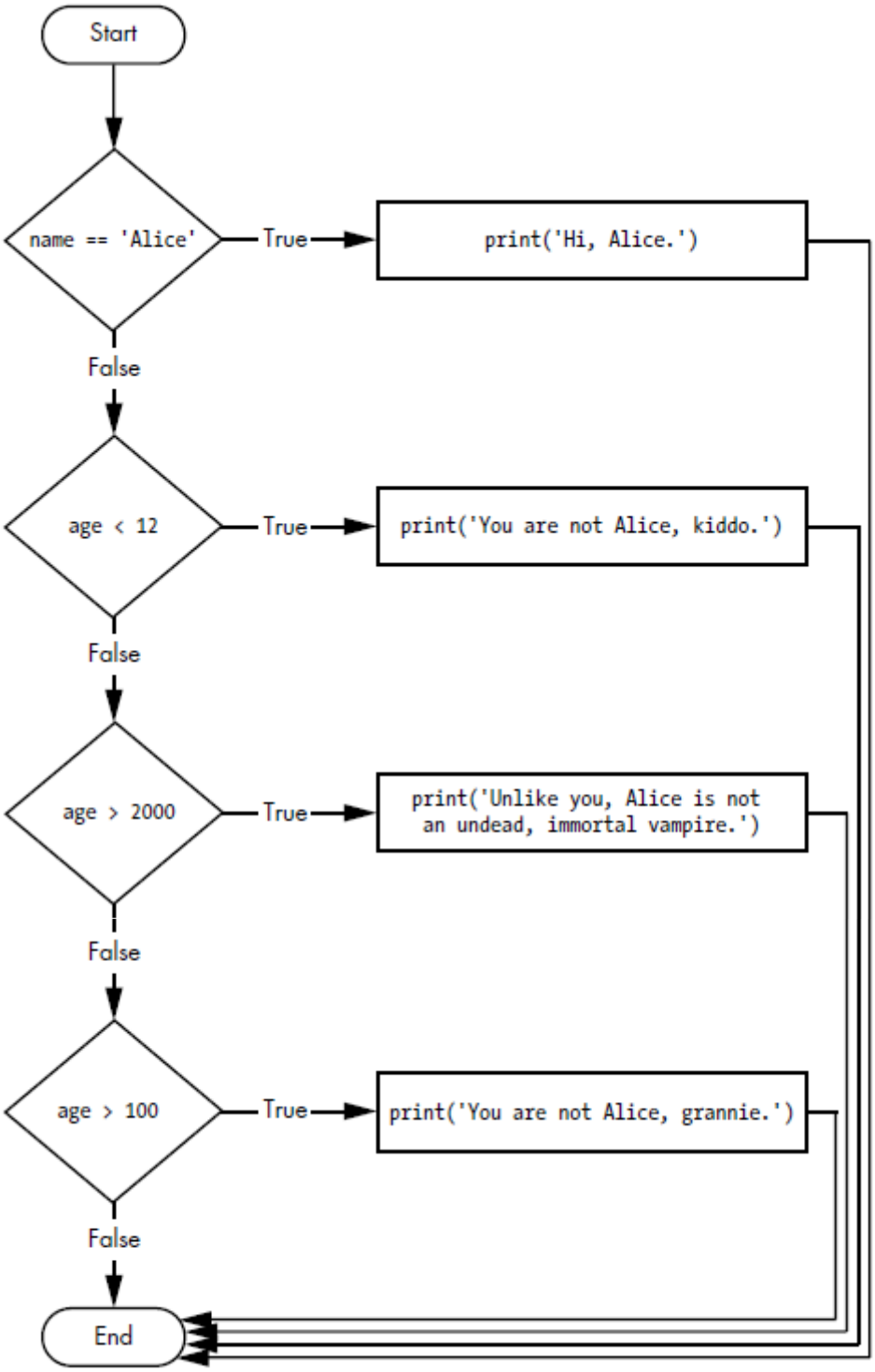
```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger')
```

La figura muestra el diagrama de flujo de este código.

SENTENCIAS ELIF.

Mientras que en las sentencias `if` y `else` solo se ejecuta una de las cláusulas (la del `if` o la del `else`), en ocasiones puede que queramos que se ejecute una cláusula de entre *muchas* posibles. La sentencia `elif` es una sentencia "else if" que siempre sigue a una sentencia `if` o a otra sentencia `elif`. La sentencia `elif` proporciona otra condición que solo se comprueba si alguna de las condiciones previas fue `False`. En Python, una sentencia `elif` siempre consiste en lo siguiente:

- La palabra reservada `elif`.
- Una condición (esto es, una expresión que se evalúa a `True` o a `False`)
- Dos puntos.
- Empezando en la siguiente línea, un bloque de código tabulado (la cláusula `elif`).



Vamos a añadir un `elif` al programa previo para ver esta sentencia en acción:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

Esta vez, comprobamos la edad del usuario, y el programa le dirá algo diferente si tiene menos de 12 años. La cláusula `elif` se ejecuta si `age < 12` es `True` y `name == 'Alice'` es `False`. Sin embargo, si ambas condiciones son `False`, entonces no se ejecuta ninguna de las cláusulas. No hay garantías de que se ejecute al menos una de las cláusulas. Cuando tenemos una cadena de sentencias `elif`, solo una o ninguna de las cláusulas se ejecutará. En cuanto una de las condiciones de las distintas sentencias se evalúa a `True`, automáticamente se ignoran el resto de cláusulas `elif`.

Por ejemplo, abre una nueva ventana en el editor de archivos y escribe el siguiente código. Guárdalo como `vampire.py`. En este último código hemos añadido dos sentencias `elif` más para hacer que el programa se dirija al usuario con diferentes respuestas en función de su edad. La figura muestra el diagrama de flujo de este programa.

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an inmortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

El orden de las sentencias `elif` en un programa es *crucial*. Vamos a reescribir el programa previo para verificar este hecho. Recuerda que una vez que el programa ha encontrado una condición que se evalúa a `True`, el resto de cláusulas `elif` se omiten automáticamente. Por consiguiente, si intercambiamos algunas de las cláusulas del programa `vampire.py` como muestra la figura, nos encontraremos con problemas. (Guarda este programa modificado como `vampire2.py`).

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
❶ elif age > 100:
    print('You are not Alice, grannie.')
elif age > 2000:
    print('Unlike you, Alice is not an inmortal vampire.')
```

Digamos que a la variable `age` se le asigna el valor `3000` antes de que este código se ejecute. Puede que pensemos que el programa responderá `'Unlike you, Alice is not an inmortal vampire.'`. Pero como en (1) la condición `age > 100` se evalúa a `True` (porque `3000` es mayor que `100`), la cadena que se imprime por pantalla es `'You are not Alice, grannie.'`, y el resto de sentencias `elif` se omiten automáticamente. Es importante recordar que, como mucho, solo se ejecutará una de las cláusulas, y que en las sentencias `elif` encadenadas, el orden es muy importante.

Opcionalmente, podríamos añadir una sentencia `else` después de la última sentencia `elif`. En este caso quedará garantizado que al menos una (y solo una) de las cláusulas se ejecutará. Si las condiciones en todos las sentencias `if` y `elif` son `False`, entonces se ejecutará la cláusula del `else`. Por ejemplo:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
     print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

En palabras, este tipo de estructura de control de flujo diría: "Si la primera condición es cierta, haz esto. En caso contrario, si la segunda condición es cierta, haz aquello. En cualquier otro caso, haz esto otro". Recuerda que en este tipo de estructuras el orden es importante. En primer lugar, siempre hay exactamente una sentencia `if`. En segundo lugar, todas las sentencias `elif` que necesitemos deben de seguir al `if`. Por último, si queremos asegurarnos de que se ejecutará al menos una cláusula, debemos cerrar la estructura con una sentencia `else`.

SENTENCIAS WHILE (BUCLES).

Con una sentencia `while` podemos hacer que un bloque de código se ejecute una y otra vez. El código dentro de la cláusula `while` se estará ejecutando repetidamente mientras que la condición del `while` siga siendo `True`.

En Python, una sentencia `while` siempre consiste en:

- La palabra reservada `while`.
- Una condición.
- Dos puntos.
- Empezando en la siguiente línea, un bloque de código tabulado (la cláusula `while`).

Como vemos, la estructura de un `while` es muy parecida a la de un `if`. La diferencia está en cómo se comporta. Al final de una cláusula `if`, la ejecución del programa continua con la siguiente instrucción tras la sentencia `if`. Pero al final de una cláusula `while`, la ejecución del programa salta de vuelta al principio de la sentencia `while`. Es por ello que a las sentencias `while` se las denomina **bucles while**, o simplemente **bucles**. Vamos a comparar el funcionamiento de una sentencia `if` y de una sentencia `while` con la misma condición y las mismas acciones basadas en esa condición. El código con la sentencia `if` es:

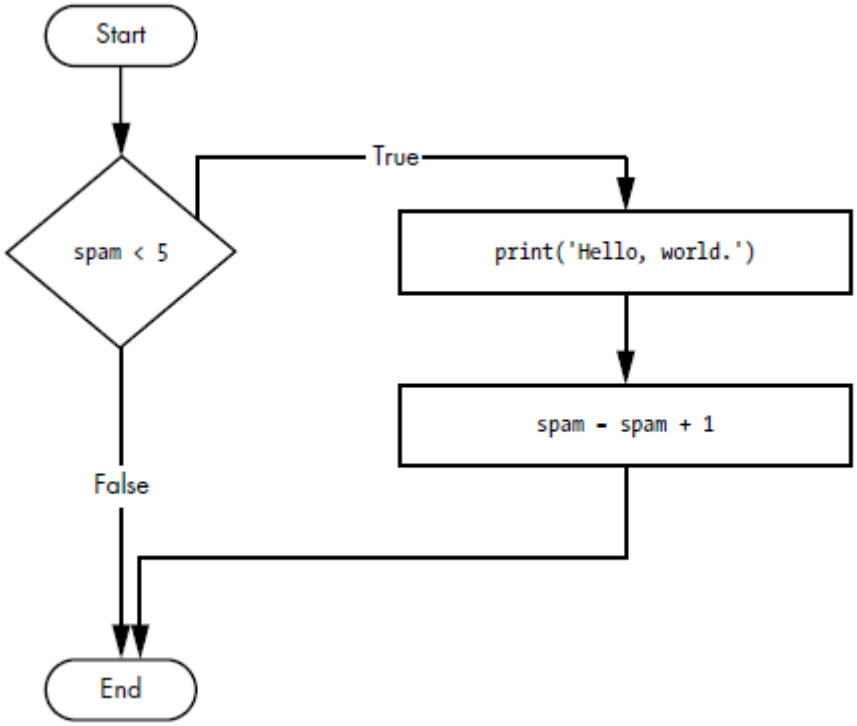
```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Por su parte, el código con la sentencia `while` es:

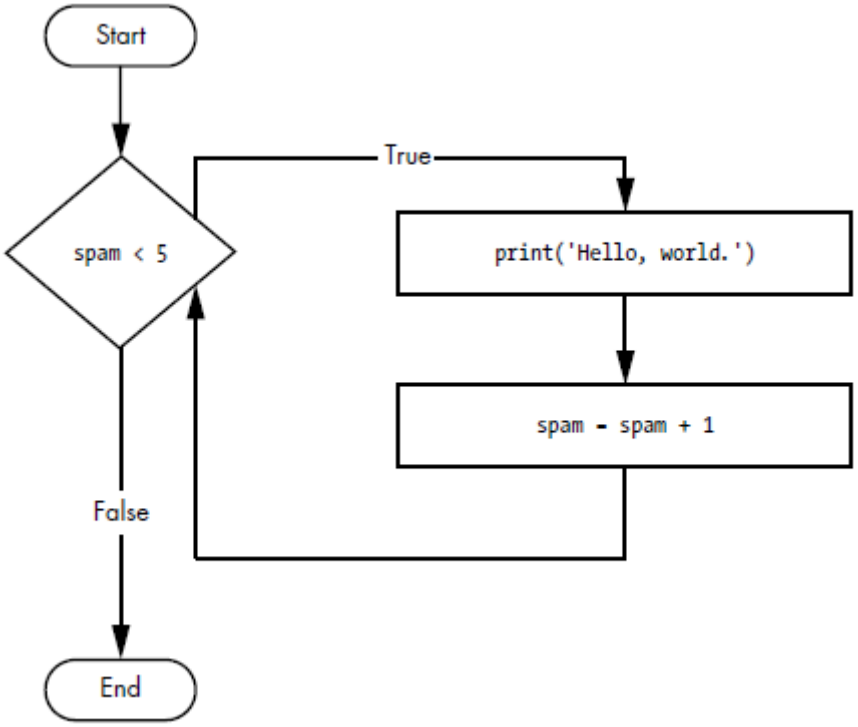
```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Ambos códigos son casi idénticos, pero cuando los ejecutamos ocurren cosas muy diferentes. Para el código con la sentencia `if`, la salida es simplemente la impresión de la cadena `'Hello, world.'` una única vez. Pero el código con la sentencia `while` imprime esa misma cadena cinco veces. Echemos un vistazo a los diagramas de flujo de ambos códigos.

El diagrama de flujo del código con el `if` es:



Y el diagrama para el código del while es:



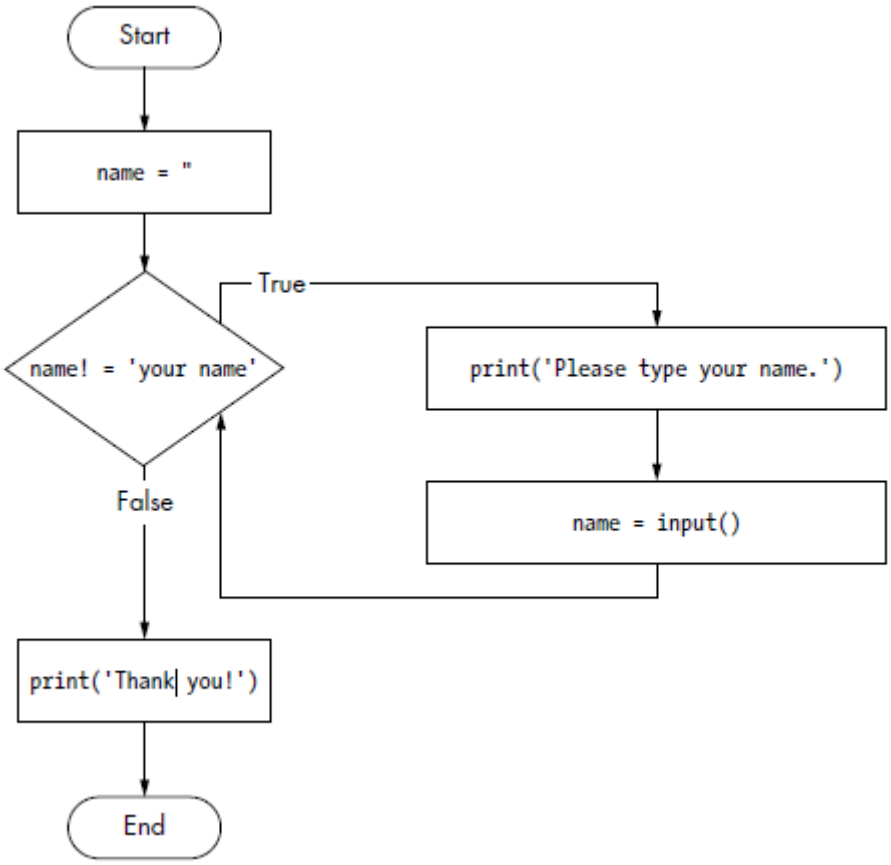
El código con la sentencia `if` comprueba la condición, e imprime `Hello, world.` una sola vez si esa condición se evalúa a `True`. Por otro lado, el código con la sentencia `while` imprime el mismo mensaje cinco veces. Deja de imprimir el mensaje tras hacerlo en cinco ocasiones porque el entero en spam se incrementa en uno al final de cada iteración del bucle, lo que significa que el bucle se ejecutará cinco veces antes de que la condición `spam < 5` se haga `False`.

En un bucle `while`, la condición siempre se ejecuta al principio de cada **iteración** (esto es, cada vez que el bucle se ejecuta). Si la condición es `True`, la cláusula se ejecuta, y a continuación, la condición vuelve a comprobarse. La primera vez que la condición se hace `False`, la cláusula `while` se omite.

Vamos a escribir un programa de ejemplo que pide constantemente que escribas, literalmente, tu nombre (`your name`). Abre una nueva ventana del editor de archivos, e inserta en siguiente código. Guárdalo como `yourName.py`.

```
❶ name = ''
❷ while name != 'your name':
      print('Please type your name.')
❸     name = input()
❹ print('Thank you!')
```

En primer lugar, en (1), el programa asigna a la variable `name` una cadena vacía. De esta forma, la condición `name != 'your name'` en (2) se evaluará a `True` y la ejecución del programa entrará en la cláusula `while`. El código dentro de esta cláusula le pide al usuario que escriba su nombre, y después, en (3), le asigna esta cadena a la variable `name`. Como ésta es la última línea de código del bloque, la ejecución retorna al principio del bucle `while` y reevalúa la condición. Si el valor en `name` no es igual a la cadena `'your name'`, la condición se evalúa a `True`, y la ejecución entra de nuevo en la cláusula `while`. Ahora bien, cuando el usuario escribe `'your name'`, la condición del bucle `while` será `'your name' != 'your name'`, y se evalúa a `False`. Como ahora la condición es `False`, la ejecución del programa ya no vuelve a entrar en la cláusula `while`, sino que la omite y continúa ejecutando el resto del programa (línea 4). La figura muestra el diagrama del flujo del programa:



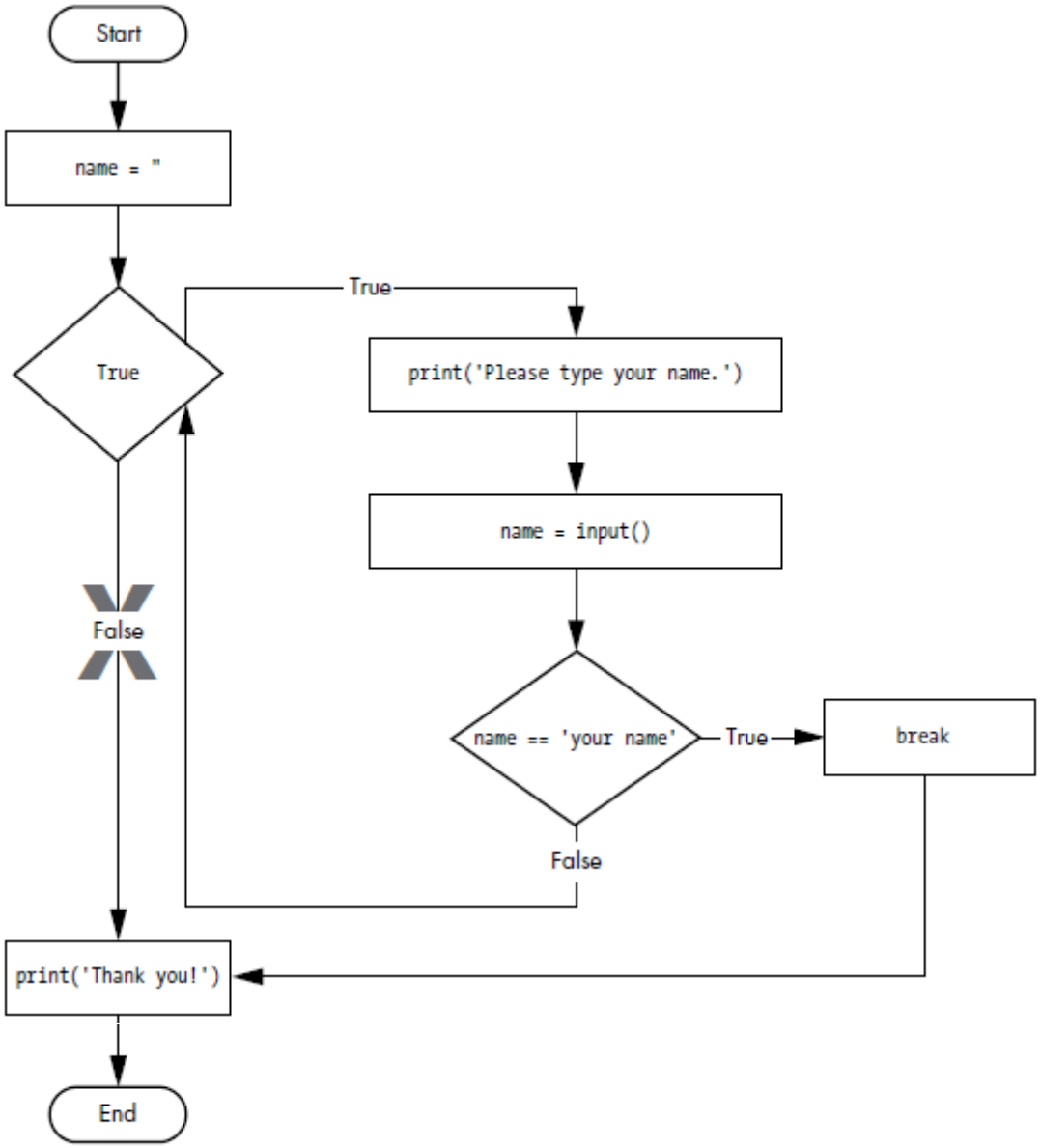
Ejecuta el programa, y cuando te pregunte, escribe algo distinto a `your name` unas cuantas veces antes de proporcionarle al programa lo que quiere.

```
Please type your name.
Al
Please type your name.
Alex
Please type your name.
%#@#%*(^&!!!
Please type your name.
your name
Thank you!
```

Si nunca escribimos `your name`, la condición del bucle `while` nunca es `False`, y el programa seguirá preguntando para siempre. En este caso, la llamada a la función `input()` le permite al usuario insertar la respuesta apropiada para hacer que el programa salga del bucle. Pero en otros programas, la condición podría no cambiar nunca, y eso sería un problema. Vamos a ver cómo podemos forzar la salida de un bucle `while`.

SENTENCIAS BREAK.

Hay un atajo para hacer que la ejecución de un programa salga de un bucle `while` antes de tiempo. Cuando la ejecución llega a una sentencia `break`, sale inmediatamente de la cláusula del bucle. En Python, una sentencia `break` solo contiene la palabra reservada `break`.



He aquí un programa que hace exactamente lo mismo que el programa previo, pero usa una sentencia `break` para escapar del bucle. Escribe este código y guárdalo como `yourName2.py`.

```
❶ while True:
      print('Please type your name.')
❷     name = input()
❸     if name == 'your name':
❹         break
❺ print('Thank you!')
```

La línea (1) crea un **bucle infinito**. Se trata de un bucle `while` cuya condición siempre es `True`. (Después de todo, la expresión `True` siempre se evalúa al valor `True`). El programa siempre entrará en el bucle y solo saldrá cuando se ejecute la sentencia `break`. (Un bucle infinito del que *nunca* se sale es un error muy común en programación).

Como antes, en (2), el programa te dice que escribas tu nombre (`your name`). Pero en este caso, mientras a ejecución del programa todavía está dentro del bucle `while`, en (3) se ejecuta una sentencia `if` para comprobar si la variable `name` es igual a `your name`. Si esta condición es `True`, en (4) se ejecutará la sentencia `break`, y la ejecución sale del bucle a la instrucción `print('Thank you!')` en la línea (5). En caso contrario se omitirá la cláusula `if` que contiene el break. En este punto, la ejecución del programa vuelve al principio de la sentencia `while`, línea (1), para comprobar de nuevo la condición. Como esta condición es simplemente el valor booleano `True`, la ejecución entra en el bucle para volver a pedirle al usuario que escriba `your name`. La figura muestra el diagrama de flujo del programa.

Ahora, ejecuta el programa y escribe el mismo texto que escribiste para el programa previo. Este nuevo programa debería funcionar de la misma forma que el anterior.

¿Atrapado en un bucle infinito?

Si alguna vez ejecutas un programa que tiene un error que le hace atascarse en un bucle infinito, presiona CTRL+C. Esta acción enviará un error `KeyboardInterrupt` al programa que lo parará inmediatamente. Para probarlo, crea el siguiente bucle infinito en el editor, y guárdalo como `infiniteLoop.py`.

```
while True:
    print('Hello world!')
```

Al ejecutar el programa, imprimirá `Hello world!` para siempre, porque la condición de la sentencia `while` siempre es `True`. En el IDLE del shell interactivo solo hay dos formas de parar este programa: presionar CTRL+C o seleccionar "Shell → Restart Shell" del menú.

SENTENCIAS CONTINUE.

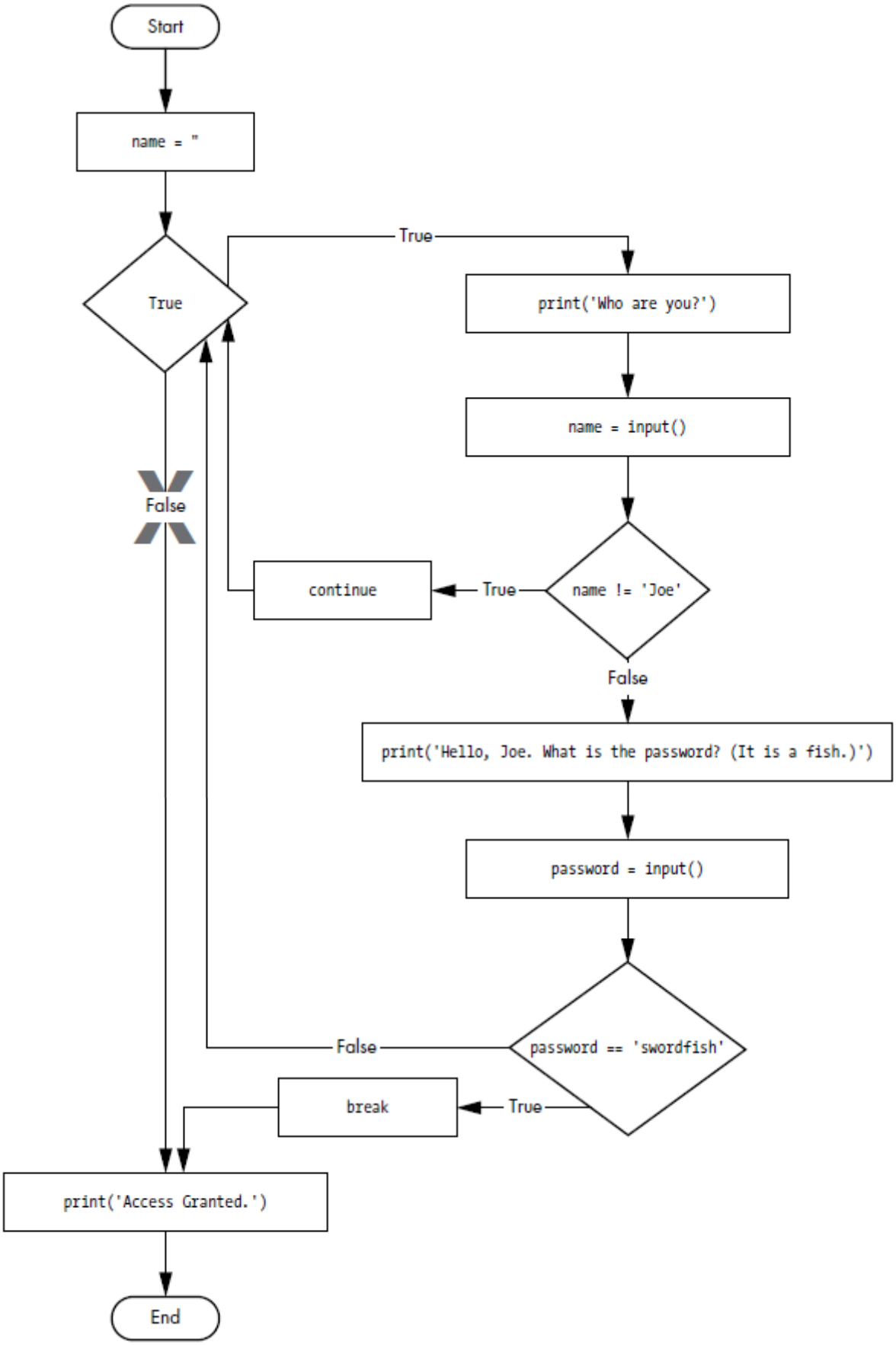
Como las sentencias `break`, las sentencias `continue` siempre se usan dentro de bucles. Cuando la ejecución llega a una sentencia `continue`, la ejecución del programa vuelve inmediatamente al principio del bucle, y reevalúa la condición del bucle. (Esto es lo que pasa también cuando la ejecución llega al final del bucle).

Vamos a usar la sentencia `continue` para escribir un programa que le pida al usuario un nombre y una contraseña. Escribe el siguiente programa en el editor de archivos y guárdalo como `swordfish.py`.

```
  while True:
      print('Who are you?')
      name = input()
❶     if name != 'Joe':
❷         continue
      print('Hello, Joe. What is the password? (It is a fish).')
❸     password = input()
      if password == 'swordfish':
❹         break
❺ print('Access granted.')
```

Si el usuario escriba un nombre que no sea `Joe`, línea (1), la sentencia `continue` en (2) hace que la ejecución del programa salte de vuelta al principio del bucle. Al reevaluar la condición, la ejecución siempre

entra en el bucle, porque la condición es simplemente el valor `True`. Cuando el usuario responde `Joe` y pasa de esa sentencia `if`, en (3) se le pide una contraseña. Si la contraseña proporcionada es `swordfish`, en (4) se ejecuta la sentencia `break`, y la ejecución sale del bucle `while` para imprimir por pantalla `Access granted`, línea (5). En otro caso, la ejecución continúa hasta el final del bucle `while`, desde donde salta atrás al principio del bucle. La figura muestra el diagrama de flujo del programa.



Ejecuta el programa y dale algunas respuestas. Hasta que no digas que eres Joe, no te pedirá una contraseña, y hasta que no introduzcas la contraseña correcta, no te dará acceso.

```
Who are you?
Sam
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish).
shark
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish).
swordfish
Access granted.
```

Valores "verdaderos" y "falsos"

Hay algunos valores pertenecientes a otros tipos de datos que las condiciones considerarán equivalentes a `True` y `False`. Al usarlos en las condiciones, los valores `0`, `0.0`, y `''` (la cadena vacía) se consideran equivalentes a `False`, mientras que todos los demás valores son equivalentes a `True`. A modo de ejemplo, observa este programa:

```
name = ''
while not name: ❶
    print('Enter your name:')
    name = input()
print('How many guests will you have?')
numOfGuests = int(input())
if numOfGuests: ❷
    print('Be sure to have enough room for all your guests.') ❸
print('Done.')
```

Si el usuario escribe una cadena vacía para `name`, la condición de la sentencia `while` será `True` (1), y el programa sigue preguntando un nombre.

(2) Si el valor de `numOfGuests` no es `0`, la condición se considera `True`, y el programa imprimirá por pantalla un recordatorio para el usuario.

Podríamos haber escrito `not name != ''` en lugar de `not name`, y `numOfGuests != 0` en vez de numOfGuests, pero haciéndolo como lo hemos hecho, el código es más fácil de leer.

BUCLES FOR Y LA FUNCIÓN RANGE().

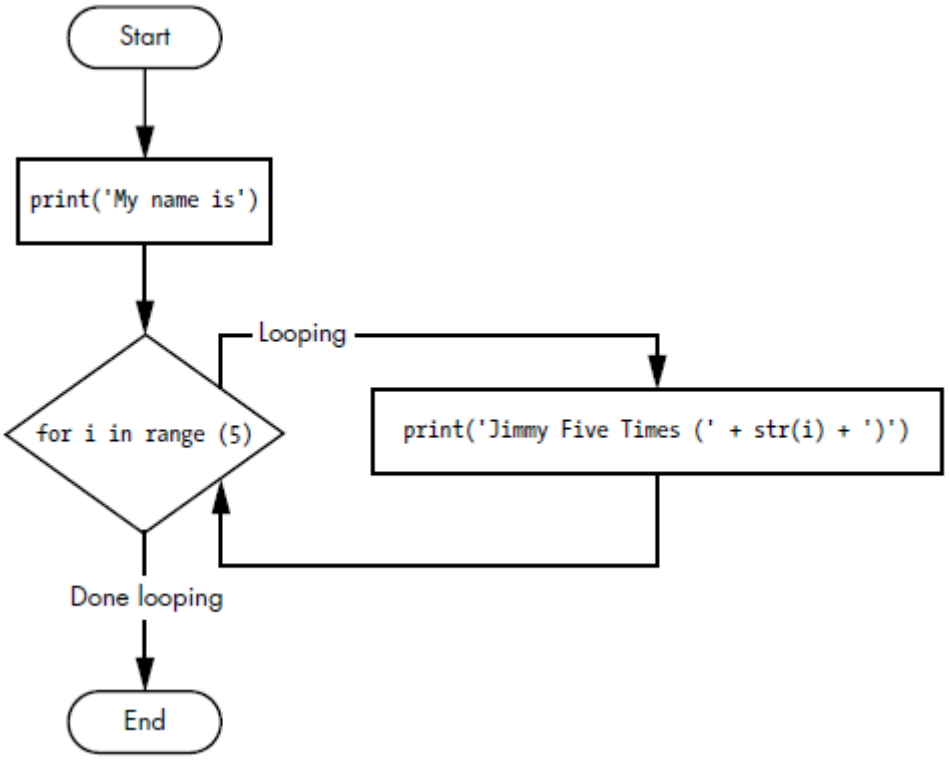
El bucle `while` continúa iterando mientras su condición es `True` (y ésta es la razón de su nombre). Pero, ¿y si queremos ejecutar un bloque de código únicamente un cierto conocido número de veces? Esto podemos hacerlo mediante un bucle `for` y la función `range()`. En Python, una sentencia `for` tiene un aspecto parecido a `for i in range(5)` : y siempre incluye lo siguiente:

- La palabra reservada `for`.
- Un nombre de variable (`i` en el ejemplo).
- La palabra reservada `in`.
- Una llamada al método `range()` pasándole hasta tres enteros
- Dos puntos.
- Empezando en la siguiente línea, un bloque de código tabulado (la cláusula `for`).

Vamos a crear un nuevo programa llamado `fiveTimes.py` para ver al bucle `for` en acción.

```
print('My name is')
for i in range(5):
    print('Alex Five Times (' +str(i)+ ')')
```

El código en la cláusula del bucle `for` se ejecuta cinco veces. La primera vez que se ejecuta, la variable `i` se ajusta a `0`, y la llamada `print()` dentro de la cláusula imprimirá `Alex Five Times (0)`. Después de que Python termine una iteración a través de todo el código dentro de la cláusula de la sentencia `for`, la ejecución retorna al principio del bucle, y la sentencia `for` incrementa el valor de `i` en una unidad. Ésta es la razón por la que la llamada `range(5)` produce cinco iteraciones a través de la cláusula, ajustando `i` inicialmente a 0, después a 1, luego a 2, a 3, y finalmente a 4. La variable `i` llegará, pero no incluirá, el entero que se le pasa a la función `range()`. La figura muestra el diagrama de flujo del programa:



Al ejecutar el programa, debería imprimir cinco veces el texto `Alex Five Times` seguido del valor actual de `i`, antes de abandonar el bucle.

```
My name is
Alex Five Times (0)
Alex Five Times (1)
Alex Five Times (2)
Alex Five Times (3)
Alex Five Times (4)
```

> NOTA: También podemos usar las sentencias `break` y `continue` dentro de los bucles `for`. La sentencia `continue` continuará hasta el siguiente valor del contador del bucle como si la ejecución del programa hubiese llegado al final del bucle y hubiese vuelto al principio. De hecho, solo podemos usar las sentencias `continue` y `break` dentro de los bucles `while` y `for`. Si intentamos usar estas sentencias en cualquier otro lado, Python nos dará un mensaje de error.

A modo de segundo ejemplo, consideremos la siguiente historia sobre el matemático Karl Friedrich Gauss. Cuando Gauss era un niño, un profesor le mandó una tarea a priori bastante dura: El profesor le dijo que sumara todos los números desde el 0 hasta el 100. El joven Gauss ideó un truco para obtener la respuesta

en unos pocos segundos, pero en nuestro caso, vamos a escribir un programa en Python que realice este cálculo con un bucle `for`.

```
total=0
for num in range(101):
    total=total+num
print(total)
```

El resultado debería ser 5050. Cuando el programa comienza, el valor de la variable `total` se fija inicialmente a 0 (línea 1). Entonces el bucle `for` (línea 2) ejecuta la instrucción `total=total+num` (línea 3) un total de 100 veces. Cuando el bucle ha terminado las 100 iteraciones, cada número entero desde 0 hasta 100 se habrá sumado a `total`. Llegados a este punto, el programa imprime `total` por pantalla (línea 4). Incluso en los ordenadores más lentos, este programa tarda menos de un segundo en completarse. (En la historia, el joven Gauss se dio cuenta de que hay 50 parejas de números que suman 100: $1 + 99$, $2 + 98$, $3 + 97$, etc., hasta $49 + 51$. Como $50 \times 100 = 5000$, lo único que queda es el 50 desemparejado, por lo que la suma de todos los números desde 0 a 100 es 5050. Ya en su juventud, Gauss era un chico muy listo).

Un bucle `while` equivalente.

Podemos usar un bucle `while` para hacer lo mismo que hace un bucle `for`; los bucles `for` simplemente son más concisos. Vamos a reescribir el programa `fiveTimes.py` usando un bucle `while` en lugar de un bucle `for`. Escribe el siguiente programa y guárdalo como `fiveTimes(2).py`:

```
print('My name is')
i=0
while i<5:
    print('Alex Five Times (' +str(i)+ ')')
    i=i+1
```

Si ejecutamos este programa, la salida debería ser la misma que obtuvimos con el bucle `for`.

Los argumentos para arrancar, parar, e incrementar la función `range()`.

Algunas funciones pueden llamarse con múltiples argumentos separados mediante comas, y la función `range()` es una de ellas. Esta opción nos permite cambiar el entero que le pasamos a `range()` para poder seguir una secuencia de enteros, incluso una que comience en un número distinto de cero.

A modo de ejemplo, escribe el siguiente programa:

```
for i in range(12,16):
    print(i)
```

El primer argumento será donde empiece la variable del bucle `for`, y el segundo argumento será el número al que llegue, sin incluirlo.

```
12
13
14
15
```

También podemos llamar a la `range()` con tres argumentos. Los dos primeros argumentos son los valores de inicio y parada, y el tercero es el **argumento de incremento**. El incremento es la cantidad en la que la variable se incrementará tras cada iteración.

```
for i in range(0,10,2):
    print(i)
```

Así pues, la llamada `range(0,10,2)` contará desde cero hasta ocho en pasos de dos.

```
0
2
4
6
8
```

La función `range()` es flexible en la secuencia de números que produce para los bucles `for`. Por ejemplo, incluso podemos usar un número negativo para el argumento de incremento para hacer que el bucle cuente hacia atrás en vez de hacerlo hacia adelante.

```
for i in range(5,-1,-1):
    print(i)
```

Si ejecutamos un bucle `for` para imprimir `i` con `range(5,-1,-1)`, deberíamos obtener una cuenta atrás desde cinco hasta cero.

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

3.7. EL MÓDULO RANDOM.

En la sección 2.7 del capítulo previo aprendimos cómo importar funciones integradas de Python mediante la sentencia `import`, e importamos el módulo `math` para acceder a algunas funciones matemáticas, como la raíz cuadrada, el seno, el coseno, etc.

Otro módulo muy útil en Python es el módulo `random`, el cual nos da acceso a la función `random.ranint()`. Escribe este código en el editor de archivos, y guárdalo como `printRandom.py`:

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

Cuando ejecutamos este programa, la salida se parecerá a lo siguiente:

```
8
7
4
3
6
```

La llamada a la función `random.randint()` se evalúa a un valor entero aleatorio entre los dos enteros que le pasemos. Como `randint()` está contenido en el módulo `random`, primero debemos escribir `random.` delante del nombre de la función para decirle a Python que busque esta función dentro del módulo `random`.

En los ejercicios de este capítulo tendrás oportunidad de usar la función `random.randint()` en varios contextos distintos.

3.8. TERMINAR UN PROGRAMA CON SYS.EXIT().

El último concepto de control de flujo que nos queda por cubrir en este capítulo es cómo terminar el programa. Esto siempre ocurre cuando la ejecución del programa llega a la última instrucción del programa. Sin embargo, podemos hacer que el programa termine, o salga, llamando a la función `sys.exit()`. Como esta función está en el módulo `sys`, debemos importar este módulo para que nuestro programa pueda usarla.

Abre una nueva ventana del editor de archivos y escribe el siguiente código. Guárdalo como `exitExample.py`:

```
import sys
while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

Ejecuta este programa en el IDLE. Este programa usa un bucle infinito que no incluye una sentencia `break` dentro de él. La única forma de terminar el programa es que el usuario escriba `exit`, lo que implicará una llamada a la función `sys.exit()`. Cuando `response` se haga igual a `exit`, el programa termina. Como el valor de la variable `response` se fija mediante la función `input()`, el usuario debe escribir `exit` para parar el programa.

3.9. UN PROGRAMA CORTO: ADIVINA EL NÚMERO.

Los ejemplos que hemos visto hasta ahora han sido útiles para presentar los conceptos básicos, pero no eran muy realistas. Ahora vamos a ver cómo encaja todo lo que hemos aprendido en un programa más completo. En esta sección vamos a mostrar cómo se escribiría un sencillo juego de "adivinar el número". Cuando ejecutemos este programa, la salida se parecerá a lo siguiente:

```
I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too high.
Take a guess.
5
Your guess is too low.
Take a guess.
7
Your guess is too low.
Take a guess.
8
Your guess is too low.
Take a guess.
9
Good job! You guessed my number in 5 guesses!
```

Escribe el siguiente código fuente en el editor de archivos, y guárdalo como `guessTheNumber.py`:

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())

    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break # This condition is the correct guess!

if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

Vamos a analizar este código línea a línea, comenzando desde el principio:

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
```

En primer lugar, y como es habitual en programación, tenemos un comentario al principio del código que explica qué es lo que hace el programa. A continuación, el programa importa el módulo `random` para poder usar la función `random.randint()` para generar el número que el usuario ha de adivinar. El valor de retorno, un número entero aleatorio entre 1 y 20, se almacena en la variable `secretNumber`.

```
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())
```

El programa de dice al usuario que ha obtenido un número secreto, y le da al usuario seis oportunidades para adivinarlo. El código que le permite al usuario insertar un número y que comprueba si es correcto está dentro de un bucle `for` que iterará un máximo de 6 veces. Lo primero que ocurre en el bucle es que el jugador inserta su conjetura. Como la función `input()` siempre devuelve una cadena, su valor de retorno se pasa directamente como argumento de la función `int()`, que convierte la cadena en un valor entero. Este valor se almacena en una variable llamada `guess`.

```
if guess < secretNumber:
    print('Your guess is too low.')
elif guess > secretNumber:
    print('Your guess is too high.')
```

Las cuatro siguientes líneas de código sirven para comprobar si el número propuesto por el usuario es mayor o menor que el número secreto. En cualquier caso, se muestra una pista por pantalla.

```
else:
    break # This condition is the correct guess!
```

Si el número propuesto por el usuario no es ni mayor ni menor que el número secreto es porque debe ser igual al número secreto, en cuyo caso queremos que la ejecución del programa salga del bucle `for`. Esto lo hacemos con la sentencia `break`.

```
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

Finalmente, Y al salir del bucle `for`, la sentencia `if` comprueba si el jugador ha adivinado correctamente el número secreto, e imprime el mensaje apropiado por pantalla (indicando también el número de intentos que ha necesitado). En caso contrario, la ejecución del programa ha abandonado el bucle no porque el usuario haya adivinado el número, sino porque ha agotado las 6 oportunidades que tenía sin conseguirlo. En ese caso, el programa muestra por pantalla un mensaje que le indica cuál era el número secreto. En ambos casos, el programa muestra una variable que contiene un valor entero (`guessesTaken` en un caso, y `secretNumber` en el otro). Como debemos concatenar estos valores enteros con unas cadenas, el programa les pasa estas variables a la función `str()`, que devuelve la forma tipo cadena de estos enteros. Ahora, se pueden concatenar estas cadenas mediante el operador `+` antes de pasarle a la llamada a la función `print()` la cadena combinada.

3.10. EJERCICIOS DEL CAPÍTULO 3.

Ejercicio 3.1. Escribe el programa que implementa el diagrama de flujo que permite decidir qué hacer si está lloviendo (ver primera página de este capítulo). Guarda el programa como `Ejer3.1.py`.

Ejercicio 3.2. Escribe un programa que le pida al su nombre y su edad. En base a la edad, el programa le indica personalmente al usuario (dirigiéndose a él por su nombre) si es mayor de edad o menor de edad. Guarda el programa como `Ejer3.2.py`.

Ejercicio 3.3. Escribe un programa que le pida al usuario un número entero cualquiera. El programa deberá determinar si ese número es par o impar, e indicarlo por pantalla. (Pista: Para saber si un número es par o impar, debes saber si es o no divisible por dos. ¿Cómo puedes comprobar si un número es divisible por otro?). Guarda el programa como `Ejer3.3.py`.

Ejercicio 3.4. Escribe un programa que imprima que le pida al usuario un número cualquiera entre el 1 y el 10. Este programa imprime por pantalla `Hello` si el valor introducido es menor o igual que 2, `Goodbye` si el valor está entre 3 y 5 (ambos inclusive), y `Congratulations` si el valor es mayor o igual que 6. Guarda el programa como `Ejer3.4.py`.

Ejercicio 3.5.
(a) Escribe un programa que muestre por pantalla todos los impares desde 1 hasta un cierto número entero introducido por el usuario. (Por ejemplo, si el usuario inserta el 8, el programa debe mostrar los impares 1, 3, 5, y 7). Guarda el programa como `Ejer3.5a.py`.
(b) Escribe un programa que muestre por pantalla los n primeros números pares, comenzando con el 0. El número n es un entero que introduce el usuario. (Por ejemplo, si el usuario inserta un 7, el programa debe proporcionar los 7 primeros números pares comenzando desde cero, esto es, 0, 2, 4, 6, 8, 10, y 12). Guarda el programa como `Ejer3.5b.py`.

Ejercicio 3.6. Escribe un programa que muestre por pantalla los 20 primeros números enteros que son impares. Además, el programa debe hacer la suma de estos 20 números impares. Guarda el programa como `Ejer3.6.py`.

Ejercicio 3.7. Escribe un programa que rellene automáticamente una quiniela de 10 partidos. Para ello, el programa debe sacar un total de 10 números aleatorios entre el 0 y el 2, donde un 0 significa empate (X), un 1 representa una victoria local, y un dos indica una victoria visitante. Guarda el programa como `Ejer3.7.py`.

Ejercicio 3.8.
(a) Programa un simulador de lanzamiento de moneda al aire. Cada vez que ejecutes el programa, éste te dará un resultado aleatorio entre `Cara` o `Cruz`.
(b) Completa el programa previo pidiéndole al usuario que adivine qué resultado va a salir. Si acierta, el programa felicita al usuario. Si falla, también se lo indica. Guarda el programa como `Ejer3.8.py`.

Ejercicio 3.9. Escribe un programa que construya una tabla de conversión de grados Celsius (columna 1) a kelvin (columna 2) y a grados Fahrenheit (columna 3). Comienza con $-270\,°C$, y va subiendo en pasos de 10 grados, hasta llegar a los $1000\,°C$. (Las fórmulas de conversión las indicamos en la sección 2.8 del capítulo 2). Para resolver este problema, te será muy útil definir una variable que almacene el incremento de temperatura entre dos temperaturas consecutivos, que en este caso es de $10\,°C$. Guarda el programa como `Ejer3.9a.py`.

Cuando hayas terminado, escribe un programa para permitirle al usuario seleccionar la temperatura inicial (que no puede ser menor que $-273\,°C$, el cero absoluto), la temperatura final (que puede ser cualquiera), y el número de filas que tendrá la tabla (lo que te fijará el incremento entre dos temperaturas consecutivas). Guarda el programa como `Ejer3.9b.py`.

Ejercicio 3.10. Secuencias numéricas.
(a) Los números de Fibonacci son una secuencia de enteros en los que cada uno de ellos es la suma de los dos enteros previos, con los dos primeros siendo 1 y 1. Así, los primeros miembros de la secuencia son $1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21$. Escribe un programa que calcule los números de Fibonacci hasta un cierto número n especificado por el usuario. Prueba tu programa para hallar los números de Fibonacci que sean menores o iguales que 1000. Guarda el programa como `Ejer3.10a.py`.
(b) Los números de Catalan C_n son una secuencia de enteros $1,\ 1,\ 2,\ 5,\ 14,\ 42,\ 132,\ \ldots$ que desempeñan un papel de importancia en mecánica cuántica y en teoría de sistemas desordenados. Estos números vienen dados por la siguiente fórmula *recursiva*:

$$C_0 = 1 \qquad C_{n+1} = \frac{4n+2}{n+2} C_n$$

Escribe un programa que imprima en orden creciente todos los números de Catalan que son menores o iguales a mil millones. Guarda el programa como `Ejer3.10b.py`.

Ejercicio 3.11. En física y matemáticas es habitual tener que evaluar sumas con grandes cantidades de términos. En ciertas situaciones, es conveniente usar un bucle para evaluar dichas sumas. Por ejemplo, suponer que queremos conocer el valor de la suma:

$$s = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{100} \equiv \sum_{k=1}^{100} \frac{1}{k}$$

, donde hemos aprovechado para presentar la notación basada en sumatorios ($\sum$), que nos permite escribir sumas de muchos términos (incluso infinitos) de forma ágil y compacta. Escribe un programa que evalúe la suma de los n primeros términos de la serie $\sum_{k=1}^{n} 1/k$, donde n es un número configurable por el usuario. Guarda el programa como `Ejer3.11.py`.

Ejercicio 3.12. Las sumas infinitas suelen aparecer de forma muy frecuente en matemáticas. Por ejemplo, las funciones seno, coseno, y exponencial pueden calcularse como una suma infinita de potencias (las llamadas **series de Taylor**):

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

, donde $3!$ ("3 factorial") es $3! = 3 \times 2 \times 1$, $5!$ ("5 factorial") es $5! = 5 \times 4 \times 3 \times 2 \times 1$, etc. El cálculo del factorial de un número arbitrario n, esto es, $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$, puede hacerse usando la función `math.factorial(n)` del módulo `math`.

Para igualar el desarrollo en serie de potencias con la función a la que representa, debemos usar un número infinito de términos (eso es lo que significan los puntos suspensivos en las fórmulas). Pero usando solo un número finito de términos, podemos obtener aproximaciones a dichas funciones. Por supuesto, cuantos más términos consideremos, mejor será la aproximación de la serie de potencias a la función real. Por ejemplo, si consideramos hasta el orden 3 (la tercera potencia), la función exponencial puede aproximarse como:

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!}$$

(a) Escribe un programa que aproxime la función exponencial para un valor de x dado hasta la potencia que indique el usuario. Por ejemplo, si el usuario indica que quiere aproximar la función exponencial en $x = 1.75$ hasta el orden 3, el programa debería calcular:

$$1 + (1.75) + \frac{(1.75)^2}{2!} + \frac{(1.75)^3}{3!}$$

Además, el programa calculará el valor *exacto* de la función exponencial para ese valor de x (en el ejemplo previo, el valor de $e^{1,75}$), y obtendrá el error absoluto cometido en la aproximación, calculando el valor absoluto de la resta del valor real y el valor aproximado. (También puedes calcular el error relativo, dividiendo el error absoluto entre el valor real). Guarda el programa como `Ejer3.12a.py`.
(b) Haz un programa para aproximar la función seno para un valor de x dado hasta la potencia que indique el usuario. (Cuidado: Observa que el desarrollo en serie del seno solo tiene potencias y factoriales impares. Observa también que los términos son positivos y negativos alternativamente). Guarda el programa como `Ejer3.12b.py`.
(c) Haz lo mismo para el coseno. Guarda el programa como `Ejer3.12c.py`.

Ejercicio 3.13. La fórmula semi – empírica de la masa.
En física nuclear, la fórmula semi – empírica de la masa es una fórmula para calcular la energía de enlace nuclear aproximada, B, de un núcleo atómico con número atómico Z y número másico A:

$$B = a_1 A - a_2 A^{2/3} - a_3 \frac{Z^2}{A^{1/3}} - a_4 \frac{(A-2Z)^2}{A} + \frac{a_5}{A^{1/2}}$$

, donde las constantes a_i están expresadas en unidades de MeV (millones de electrón voltios), y sus valores son $a_1 = 15{,}67,\ a_2 = 17{,}23,\ a_3 = 0{,}75,\ a_4 = 93{,}2$, y:

$$a_5 = \begin{cases} 0 & si\ A\ es\ impar \\ 12{,}0 & si\ A\ y\ Z\ son\ ambos\ pares \\ -12{,}0 & si\ A\ es\ par\ y\ Z\ es\ impar \end{cases}$$

(a) Escribe un programa que reciba como datos de entrada los valores de A y de Z, e imprima la energía de enlace del átomo correspondiente. Usa tu programa para hallar la energía de enlace de un átomo con $A = 58$ y $Z = 28$. (Pista: La respuesta correcta es alrededor de $490\ MeV$).
(b) Modifica tu programa para que imprima, no la energía de enlace total B, sino la energía de enlace por nucleón, B/A.
(c) Ahora modifica tu programa para que tome como dato de entrada únicamente el número atómico Z, y a continuación, recorra todos los valores de A desde $A = Z$ hasta $A = 3Z$, para hallar aquel valor de A que tiene la mayor energía de enlace por nucleón. La respuesta nos indicará cuál es el núcleo más estable para el número atómico dado. Haz que tu programa imprima el valor de A del núcleo más estable, así como el valor de su energía de enlace por nucleón
(d) Modifica de nuevo tu programa para que, en vez de recibir como entrada el valor de Z, recorra todos los valores de Z desde 1 hasta 100, e imprima el valor más estable de A para cada uno de ellos. ¿Para qué valor de Z se tiene la máxima energía por nucleón? (En el mundo real, la respuesta es $Z = 28$, que es el níquel. Deberías obtener que la fórmula semi – empírica proporciona una respuesta aproximadamente correcta, pero no exactamente correcta).

4. FUNCIONES.

4.1. INTRODUCCIÓN A LAS FUNCIONES.

De los capítulos previos, ya estamos familiarizados con las funciones `print()`, `input()`, y `len()`. Python incluye muchas más funciones integradas como éstas pero también nos permite escribir nuestras propias funciones. Una **función** es como un mini-programa dentro de un programa. Para entender mejor cómo funcionan las funciones, vamos a crear una. Escribe el siguiente programa en el editor de archivos y guárdalo como `helloFunc.py`:

```
❶ def hello():
❷     print('Hi!')
      print('Hi!!!')
      print('Hello there.')

❸ hello()
  hello()
  hello()
```

La primera línea (1) es una sentencia `def`, que define una función llamada `hello()`. El código en el bloque que sigue a la sentencia `def` es el cuerpo de la función (2). Este código se ejecuta cuando se llama a la función, no cuando la función se define por primera vez.

En (3), las líneas `hello()` tras la definición de la función son las llamadas que hace el **programa principal** a esa función. En Python, una llamada a una función consiste simplemente en el nombre de la función seguido de paréntesis, posiblemente pasándole uno o más argumentos entre los paréntesis. Cuando la ejecución del programa llega a estas llamadas, el programa salta a la primera línea de la definición de la función, y empieza a ejecutar su código. Cuando llega al final de la función, la ejecución retorna a la línea que llamó a la función, y continúa avanzando por el código como antes. (Observar que, en Python, es tradición escribir la definición de las funciones al principio del programa que las utiliza). Como el programa llama a `hello()` tres veces, el código dentro de la función `hello()` se ejecuta tres veces. Cuando ejecutamos este programa, la salida que obtenemos es:

```
Hi!
Hi!!!
Hello there.
Hi!
Hi!!!
Hello there.
Hi!
Hi!!!
Hello there.
```

Uno de los propósitos principales de las funciones es agrupar el código que debe ejecutarse múltiples veces. Si no hubiésemos definido esta función, tendríamos que copiar y pegar este código cada vez que quisiésemos ejecutarlo, y el programa se parecería a éste:

```
print('Hi!')
print('Hi!!!')
print('Hello there.')
print('Hi!')
print('Hi!!!')
print('Hello there.')
print('Hi!')
print('Hi!!!')
print('Hello there.')
```

En general, siempre querremos evitar duplicar código, porque si alguna vez necesitamos modificarlo (por ejemplo, si encontramos un error que debemos corregir), tendremos que cambiar el código en todos los sitios donde lo copiamos. De hecho, conforme adquiramos más experiencia en programación, habitualmente nos encontraremos eliminando código duplicado de programas que escribimos con anterioridad.

4.2. SENTENCIAS DEF CON PARÁMETROS.

Cuando llamamos a las funciones `print()` o `len()`, les pasamos valores (a los que en este contexto llamamos *argumentos*) escribiéndolos entre los paréntesis de la función. También podemos definir funciones propias que acepten argumentos. Escribe el siguiente ejemplo en el editor de archivos y guárdalo como `helloFunc2.py`:

```
❶ def hello(name):
❷     print('Hello ' + name)

❸ hello('Alice')
  hello('Bob')
```

Al ejecutar este programa, la salida que obtenemos es:

```
Hello Alice
Hello Bob
```

(1) En este programa, la definición de la función `hello()` tiene un parámetro llamado `name`. Un **parámetro** es una variable en la que se almacena un argumento cuando se llama a una función. La primera vez que se llama a la función `hello()`, se hace con el argumento '`Alice`' (3). La ejecución del programa entre en la función, y el valor de la variable name se fija automáticamente a '`Alice`', que es lo que se imprime con la sentencia `print()` en (2).

Una cosa que debemos recordar sobre los parámetros es que el valor almacenado en un parámetro se borra cuando el programa retorna de la función. Por ejemplo, si añadiésemos un `print(name)` después de `hello('Bob')` en el programa previo, el programa nos daría un error `NameError` porque no habría una variable llamada `name`. Esta variable sería destruida después de que la llamada a la función `hello('Bob')` retornase, por lo que la llamada `print(name)` se referiría a una variable `name` que no existe.

Esto es similar a la forma en la que las variables se borran cuando el programa termina. Más adelante hablaremos de la razón por la que ocurre esto, cuando discutamos qué es el ámbito local de una función.

4.3. VALORES DE RETORNO Y SENTENCIAS RETURN.

Cuando llamamos a la función `len()` y le pasamos un argumento como '`Hello`', la llamada a la función se evalúa al valor entero `5`, que es la longitud de la cadena que le hemos pasado. En general, el valor al que se evalúa la llamada a una función se le denomina **valor de retorno** de la función.

Al crear una función mediante una sentencia `def`, podemos especificar cuál debería ser el valor de retorno usando una sentencia `return`. Una sentencia `return` consiste en lo siguiente:

- La palabra reservada `return`.
- El valor o expresión que la función debería devolver.

Cuando usamos una expresión con una sentencia `return`, el valor de retorno es aquel al que esa expresión se evalúa. A modo de ejemplo, el siguiente programa define una función para calcular el valor promedio de

dos números enteros que se le pasan como argumentos. Escribe este código en el editor de archivos y guárdalo como `average2number.py`:

```
def average(num1, num2):
    return (num1 + num2)/2

print('Gime me an integer number:')
a = int(input())
print('Gime me a second integer number:')
b = int(input())
ave = average(a, b)
print('The average of these two numbers is:')
print(ave)
```

Otra posibilidad para la definición de la función habría sido la siguiente:

```
def average(num1, num2):
    ans =(num1 + num2)/2
    return ans
```

Como segundo ejemplo, el siguiente programa define una función que devuelve una cadena distinta dependiendo de qué número le pasemos como argumento. Escribe este código en el editor de archivos y guárdalo como `magic8Ball.py`:

```
❶ import random

❷ def getAnswer(answerNumber):
❸     if answerNumber == 1:
          return 'It is certain'
      elif answerNumber == 2:
          return 'It is decidely decidedly so'
      elif answerNumber == 3:
          return 'Yes'
      elif answerNumber == 4:
          return 'Reply hazy try again'
      elif answerNumber == 5:
          return 'Ask again later'
      elif answerNumber == 6:
          return 'Concentrate and ask again'
      elif answerNumber == 7:
          return 'My reply is no'
      elif answerNumber == 8:
          return 'Outlook not so good'
      elif answerNumber == 9:
          return 'Very doubtful'

❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

Nada más comenzar el programa, Python importa el módulo `random` (1). A continuación (2), se define la función `getAnswer()`. Como aquí solo se estamos definiendo la función (no la estamos llamando), la ejecución se salta el código contenido dentro de ella. En (4), el programa hace una llamada a la función `random.randint()` pasándole dos argumentos, 1 y 9. Esta llamada se evalúa a un entero aleatorio entre 1 y 9 (ambos incluidos), y este valor se guarda en una variable llamada `r`.

En (5) el programa hace una llamada a la función `getAnswer()` pasándole `r` como argumento. Entonces, la ejecución del programa salta al principio de la función `getAnswer()`, línea (3), y el valor `r` se almacena en un parámetro llamado `answerNumber`. A continuación, dependiendo del valor que tome `answerNumber`, la

función devuelve una de las muchas cadenas posibles. La ejecución del programa retorna a la línea (5) donde se hizo la llamada a la función `getAnswer()`. La cadena devuelta se asigna a una variable llamada `fortune`, que a continuación, en (6), se le pasa como argumento a la función `print()` para que la muestre por pantalla.

Notar que, como podemos pasar los valores de retorno como argumentos de la llamada a otra función, en nuestro programa podríamos haber acortado las últimas tres líneas:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

, en una sola línea:

```
print(getAnswer(random.randint(1,9)))
```

Recuerda: Las expresiones están compuestas de valores y operadores. Podemos usar una llamada a una función dentro de una expresión porque esa llamada se evalúa a su valor de retorno.

4.4. EL VALOR NONE.

En Python hay un valor llamado `None`, que representa la usencia de un valor. `None` es el único valor posible del tipo de datos `NoneType`. (Otros lenguajes de programación llaman a este valor `null`, `nil`, o `undefined`). Como los valores Booleanos `True` y `False`, `None` debe escribirse con N mayúscula.

Este valor sin valor puede ser de utilidad cuando necesitemos almacenar algo que no deba de confundirse con el valor real que pueda tomar una variable. Por ejemplo, un lugar donde se usa `None` es en el valor de retorno de la función `print()`. La función `print()` imprime un valor por pantalla, pero no necesita devolver nada de la forma que los hacen las funciones `len()` o `input()`. Pero como todas las llamadas a funciones deben evaluarse a un valor de retorno, la función `print()` devuelve `None`. Para ver cómo funciona esto, escribe el siguiente código en el shell interactivo:

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

En segundo plano, Python añade un `return None` al final de cualquier definición de una función que carezca de una sentencia `return`. Esto es similar a la forma en la que un bucle `while` o un bucle `for` terminan de forma implícita al usar una sentencia `continue`. Además, si usamos una sentencia `return` sin un valor (esto es, si en la sentencia `return` solo aparece la palabra reservada `return`), la función devuelve un `None`.

A modo de ejemplo, vuelve a escribir el programa que calcula el promedio de dos números, pero esta vez borra la sentencia `return`. El programa queda como:

```
def findAverage(num1, num2):
    ans =(num1 + num2)/2

print('Gime me an integer number:')
a = int(input())
print('Gime me a second integer number:')
b = int(input())
ave = findAverage(a, b)
print('The average of these two numbers is:')
print(ave)
```

Si ejecutamos el programa, la salida será similar a esta:

```
Gime me an integer number:
8
Gime me a second integer number:
10
The average of these two numbers is:
None
>>>
```

En efecto, nuestra función no incluye una sentencia `return`, por lo que el valor que devuelve es `None`, lo que representa la usencia de un valor devuelto. Obviamente, el programa no funciona de forma correcta.

4.5. ARGUMENTOS CLAVE Y FUNCIÓN PRINT().

La mayoría de los argumentos se identifican mediante su posición en la llamada a una función. Un ejemplo lo tenemos en el programa `average2number.py` que escribimos en la sección previa: El programa principal definía dos variables enteras `a` y `b` que pasaba como argumentos a la función `findAverage(num1, num2)` mediante la llamada `ave = findAverage(a, b)`. El argumento `a` se almacena en el parámetro `num1` y el argumento `b` en el parámetro `num2`, simplemente por correspondencia posicional. Otro ejemplo es la llamada `random.randint(1, 10)`, la cual es diferente de la llamada `random.randint(10, 1)`. La primera llamada devolverá un entero aleatorio entre 1 y 10, porque el primer argumento es el límite inferior del rango y el segundo el límite superior, mientras que la segunda llamada producirá un error. (Pruébalo tú mismo en el Shell interactivo).

Sin embargo, los **argumentos clave** (*keyword arguments* o *kwargs*) se identifican mediante la palabra clave que se pone delante de ellos en la llamada a la función. Los argumentos clave suelen usarse para los parámetros opcionales. Por ejemplo, la función `print()` tiene los parámetros opcionales `end` y `sep`, que sirven, respectivamente, para especificar que debería imprimirse al final de la cadena que le hemos pasado como argumento, y entre las múltiples cadenas que le podemos pasar (para separarlas).

Si ejecutamos el siguiente programa:

```
print('Hello')
print('World')
```

, la salida será:

```
Hello
World
```

Las dos cadenas aparecen en líneas separadas porque la función `print()` añade automáticamente un carácter de cambio de línea (retorno de carro) al final de la cadena que se le pasa como argumento. Sin embargo, podemos fijar el valor del argumento clave `end` para cambiar este carácter por otra cadena distinta. Por ejemplo, si el programa fuese:

```
print('Hello', end='')
print('World')
```

, la salida sería en este caso:

```
HelloWorld
```

La salida se imprime en una sola línea porque ya no se cambia de línea después de mostrar '`Hello`'. En lugar de eso, se imprime una cadena vacía. Esto que hemos hecho es útil cuando queremos deshabilitar el cambio de línea al final de cada llamada a la función `print()`.

De forma similar, cuando le pasamos múltiples cadenas a un `print()`, la función las separa automáticamente con un único espacio. Escribe lo siguiente en el shell interactivo:

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

Pero podríamos cambiar la cadena de separación por defecto pasándole a la función el argumento clave `sep`. Por ejemplo, inserta el siguiente código en el shell interactivo:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

También podemos añadir argumentos clave a nuestras propias funciones, pero antes de hacerlo, necesitamos conocer los tipos de datos *lista* y *diccionario*, de los que hablaremos en próximos capítulos. Por ahora, es suficiente con saber que algunas funciones tienen argumentos claves opcionales que pueden especificarse cuando se llama a la función.

4.6. ÁMBITOS LOCAL Y GLOBAL.

Se dice que los parámetros y variables cuyo valor se asigna dentro de una función existen en el **ámbito local** de la función. Por el contrario, se dice que las variables cuyo valor se asigna fuera de las funciones de un programa existen en el **ámbito global** de ese programa. Una variable que existe en el ámbito local de una función se denomina **variable local**, mientras que una variable que existe en el ámbito global del programa se denomina **variable global**. Toda variable debe ser de uno u otro tipo, no puede ser local y global simultáneamente.

Para entender qué significa todo esto, podemos imaginar que el ámbito es una especie de contenedor de variables. Cuando un ámbito se destruye, todas las variables almacenadas en ese ámbito se borran. Sólo hay un ámbito global, y éste se crea cuando arranca nuestro programa. Cuando el programa termina, el ámbito global se destruye, y todas sus variables se borran. Si esto no ocurriese, la siguiente vez que ejecutásemos el programa, las variables recordarían los valores que tenían la última vez que ejecutamos el programa.

Por otro lado, un ámbito local se crea siempre que se llama a una función. Todas las variables cuyo valor se asigne dentro de una función existen en el ámbito local de esa función. Cuando una función retorna, su ámbito local se destruye, y todas sus variables se borran. La siguiente vez que llamemos a esta función, las variables locales no recordarán los valores que almacenaban la última vez que se llamó a la función.

Los ámbitos son importantes por varias razones:

- El código en el ámbito global no puede usar variables locales.
- Sin embargo, el ámbito local puede acceder a las variables globales (ya veremos cómo).
- El código dentro del ámbito local de una función no puede usar variables de los ámbitos locales de otras funciones.
- Podemos usar el mismo nombre para variables diferentes si pertenecen a ámbitos distintos. Es decir, puede haber una variable local llamada `spam` y una variable global también llamada `spam`.

La razón por la que Python tiene diferentes ámbitos (en vez de hacer que todas las variables sean globales) es que, cuando las variables se ven modificadas por el código de una función a la que se ha llamado, la función interactúa con el resto del programa únicamente mediante sus parámetros y su valor de retorno. Esto reduce drásticamente el número de líneas de código que pueden estar causando un malfuncionamiento (bug). Si nuestro programa solo tuviese variables globales y causase un malfuncionamiento debido a que una variable toma un valor erróneo, sería muy difícil detectar en qué parte del programa se le asignó ese valor erróneo. Podría haber ocurrido en cualquier lugar del programa (y nuestro programa podría tener cientos o miles de líneas de código). Sin embargo, si el error se debe a una variable local con un valor erróneo,

sabemos que el código que le asignó ese valor erróneo está en la función a la que pertenece esa variable local.

A modo de ejemplo, vamos a introducir adrede un bug en el programa `average2number.py`. Escribe lo siguiente en el editor de archivos (¿puedes ver el error que hemos introducido?):

```
def findAverage(num1, num2):
    ans = num1 + num2 / 2
    return ans

print('Gime me an integer number:')
a = int(input())
print('Gime me a second integer number:')
b = int(input())
ave = findAverage(a, b)
print('The average of these two numbers is:')
print(ave)
```

Al ejecutar el programa, la salida será similar a la mostrada:

```
Gime me an integer number:
7
Gime me a second integer number:
9
The average of these two numbers is:
11.5
```

Esta salida es evidentemente errónea, porque la media de dos números enteros no puede ser mayor que ambos. ¿Dónde está el error? En este caso, es fácil acotar la zona del programa donde se produjo el malfuncionamiento. Evidentemente, el error está en la función que calcula la media, y es allí donde debo buscarlo. En efecto, el error ha sido olvidar poner la suma de los dos números entre paréntesis, para efectuarla antes que la división entre dos.

Este ejemplo es solo una muestra. Imaginemos un programa con miles de líneas, donde no hay funciones y todas las variables son globales. Tratar de buscar el origen de un malfuncionamiento en un programa así sería una tarea casi imposible. Ésta es precisamente una de las razones por la que existen los ámbitos locales y las funciones: Las funciones nos permiten compartimentar y asilar bloques de código, para que la tarea de localizar malfuncionamientos en los programas sea más sencilla.

Aunque usar variables globales en programas de tamaño reducido es una práctica aceptable, no es un buen hábito acostumbrarse a usar variables globales conforme nuestros programas se hacen más y más grandes.

LAS VARIABLES LOCALES NO PUEDEN USARSE EN EL ÁMBITO GLOBAL.

Consideremos el siguiente programa, que contiene un error:

```
def spam():
    eggs = 31337

spam()
print(eggs)
```

Si ejecutamos este programa, la salida se parecerá a esto:

```
Traceback (most recent call last):
  File "C:/Users/Alejandro/Desktop/1.py", line 5, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

El error ocurre porque la variable `eggs` solo existe en el ámbito local que se crea cuando el programa llama a la función `spam()`. Después de que la ejecución retorne de la función `spam()`, ese ámbito local se destruye, y ya no hay una variable llamada `eggs`. Por lo tanto, cuando nuestro programa intenta ejecutar la instrucción `print(eggs)`, Python nos da un mensaje de error que indica que `eggs` no está definida. Si nos paramos a pensarlo, esto tiene sentido: Cuando la ejecución de un programa está en el ámbito global no existe ningún ámbito local, por lo que no puede haber variables locales. Ésta es la razón por la que solo pueden usarse variables globales en el ámbito global.

UN ÁMBITO LOCAL NO PUEDE USAR VARIABLES DE OTROS ÁMBITOS LOCALES.

Siempre que se llama a una función se crea un nuevo ámbito local, incluyendo la situación en la que una función llama a otra función. Consideremos este programa:

```
  def spam():
❶     eggs = 99
❷     bacon()
❸     print(eggs)

  def bacon():
      ham = 101
❹     eggs = 0

❺ spam()
```

Nada más comenzar, el programa principal llama a la función `spam()`, línea (5), y se crea un ámbito local. Dentro de la función, la variable local `eggs` se fija a 99. A continuación, y dentro de la función, se llama a la función `bacon()`, línea (2), y se crea un segundo ámbito local. Notar que pueden existir varios ámbitos locales al mismo tiempo. En este nuevo ámbito local, la variable `ham` se fija a 101, y también se crea una variable local `eggs` (que es diferente a la variable `eggs` del ámbito local de `spam()`) y se fija a 0, línea (4).

Cuando la función `bacon()` retorna, su ámbito local se destruye. La ejecución del programa continúa en la función `spam()` para imprimir el valor de `eggs`, línea (3), y como el ámbito local de la llamada a `spam()` todavía existe aquí, la variable `eggs` almacena el valor 99, y ese será el valor que el programa imprimirá.

La conclusión es que las variables locales de una función están completamente separadas de las variables locales de otra función.

LOS ÁMBITOS LOCALES PUEDEN ACCEDER A LAS VARIABLES GLOBALES.

Considera el siguiente programa:

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

Como en la función `spam()` no hay un parámetro llamado `eggs`, ni tampoco un código que le asigne a `eggs` un valor, cuando la variable `eggs` se usa en `spam()`, Python la considera una referencia a la variable global `eggs`. Ésta es la razón por la que se imprime el valor 42 cuando ejecutamos el programa previo.

VARIABLES LOCALES Y GLOBALES CON EL MISMO NOMBRE.

Normalmente evitaremos usar variables locales que tengan el mismo nombre que una variable global u otra variable local de otro ámbito. Pero esta práctica es perfectamente legal en Python. Para ver lo que ocurre, escribe el siguiente código en el editor de archivos y guárdalo como `sameName.py`:

```
  def spam():
❶     eggs = 'spam local'
      print(eggs) # prints 'spam local'

  def bacon():
❷     eggs = 'bacon local'
      print(eggs) # prints 'bacon local'
      spam()
      print(eggs) # prints 'bacon local'

❸ eggs = 'global'
  bacon()
  print(eggs) # prints 'global'
```

Al ejecutar este programa, la salida que obtenemos es la siguiente:

```
bacon local
spam local
bacon local
global
```

En realidad, hay tres variables distintas en este programa, pero todas ellas se llaman `eggs`. Las variables son las siguientes:
(1) Una variable llamada `eggs` que existe en el ámbito local que se crea al llamar a la función `spam()`.
(2) Una variable llamada `eggs` que existe en el ámbito local que se crea al llamar a la función `bacon()`.
(3) Una variable llamada `eggs` que existe en el ámbito global.

Como estas tres variables tienen el mismo nombre, puede resultar difícil controlar cuál de ellas se está usando en un momento dado. Esta es la razón por la que deberíamos evitar usar el mismo nombre de variable en diferentes ámbitos.

4.7. LA SENTENCIA GLOBAL.

Si necesitamos modificar una variable global desde el código dentro de una función, debemos usar una sentencia `global`. Si tenemos una línea de código como `global eggs` al principio de una función, esta instrucción le dice a Python: "En esta función, `eggs` se refiere a la variable global, así que no crees una variable local con este nombre". Por ejemplo, escribe el siguiente código en el editor de archivos y guárdalo como `sameName2.py`:

```
  def spam():
❶     global eggs
❷     eggs = 'spam'

  eggs = 'global'
  spam()
  print(eggs)
```

Cuando ejecutemos este programa, la llamada final a la función `print()` dará la siguiente salida:

```
spam
```

Como al principio de `spam()` la variable `eggs` se declara como global, cuando `eggs` se fija al valor '`spam`' , esta asignación se hace en el ámbito global. No se crea una variable local llamada `eggs`.

Hay cuatro reglas para decidir si una variable está en un ámbito local o en el ámbito global:

1) Si una variable se está usando en el ámbito global (esto es, fuera de todas las funciones), entonces siempre se trata de una variable global.
2) Si en una función hay una sentencia `global` para una variable, se trata de una variable global.
3) En otro caso, si la variable se usa en una sentencia de asignación dentro de una función, es una variable local.
4) Pero si la variable no se usa en una sentencia de asignación, es una variable global (sin embargo, ver comentario más abajo).

Para hacernos una idea más precisa de estas reglas, veamos un ejemplo. Escribe este código en el editor de archivos y guáralo como `sameName3.py`:

```
   def spam():
❶      global eggs # this is the global
       eggs = 'spam'

   def bacon():
❷      eggs = 'bacon' # this is a local

   def ham():
❸      print(eggs) # this is the global

   eggs = 42 # this is the global
   spam()
   print(eggs)
```

(1) En la función `spam()`, `eggs` es una variable global, porque hay una sentencia `global` para `eggs` al principio de la función. (2) En la función `bacon()`, `eggs` es una variable local, porque hay una sentencia de asignación para ella en el cuerpo de la función. (3) En la función `ham()`, `eggs` es una variable global, porque dentro de la función no hay una sentencia de asignación o una sentencia `global` para esa variable. Si ejecutamos el programa `sameName3.py`, la salida que obtendremos será la siguiente:

```
spam
```

Dentro de una función, una variable siempre será o bien local o bien global. No hay forma posible de que el código de una función pueda usar una variable local llamada `eggs` y luego, en esa misma función, use la variable global `eggs`.

> NOTA: Es importante recordar que si alguna vez queremos modificar el valor almacenado en una variable global desde una función, debemos usar una sentencia `global` sobre esa variable.

Si intentamos usar una variable local dentro de una función antes de haberle asignado un valor (como en el programa a continuación), Python nos dará un mensaje de error. Escribe el siguiente código en el editor de archivos y guárdalo como `sameName4.py`:

```
   def spam():
       print(eggs) # ERROR!
❶      eggs = 'spam local'

❷  eggs = 'global'
   spam()
```

Si ejecutamos este programa, obtenemos un mensaje de error similar a éste:

```
Traceback (most recent call last):
  File "C:/Users/Usuario/Dropbox/PYTHON/PROGRAMAS/4 sameName4.py", line 6, in <module>
    spam()
  File "C:/Users/Usuario/Dropbox/PYTHON/PROGRAMAS/4 sameName4.py", line 2, in spam
    print(eggs) # ERROR!
UnboundLocalError: local variable 'eggs' referenced before assignment
```

Este error ocurre porque Python ve que en (1) hay una sentencia de asignación para la variable `eggs` en la función `spam()`, y por lo tanto, considera que `eggs` es una variable local. Pero como la instrucción `print(eggs)` se ejecuta antes de que a `eggs` se le asigne un valor, la variable local `eggs` no existe. Python no recurre a la variable local `eggs` definida en (2).

Funciones como "cajas negras".

A menudo, todo lo que necesitamos saber de una función son sus entradas (los parámetros) y el valor de salida. En esos casos, no tenemos que preocuparnos por la forma en la que funciona el código de esa función. Cuando pensamos en las funciones a este alto nivel, es muy habitual decir que estamos tratando a la función como una "caja negra".

Esta idea es fundamental en la programación moderna. Más adelante veremos varios módulos con funciones que fueron escritas por otras personas. Aunque podemos echar un vistazo al código de estas funciones, realmente no necesitamos saber cómo funcionan para poder usarlas. Y como aquí hemos recomendado escribir funciones sin variables globales, normalmente no tendremos que preocuparnos por la forma en la que el código de la función interactúa con el resto del programa.

4.8. MANEJO DE EXCEPCIONES.

Con lo que sabemos hasta ahora, cuando nuestro programa genera un error o **excepción**, falla el programa entero. Evidentemente, no queremos que esto ocurra; lo que queremos es que el programa detecte los errores, los gestione, y continúe ejecutándose normalmente. A modo de ejemplo, consideremos el siguiente programa, que incluye un error de tipo "división por cero". Abre una nueva ventana del editor de archivos, escribe el siguiente código, y guárdalo como `zeroDivide.py`:

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Aquí hemos definido una función llamada `spam`, le hemos dado un parámetro, y a continuación hemos mostrado por pantalla el valor de esa función con varios parámetros para ver qué ocurre. Ésta es la salida que obtenemos cuando ejecutamos este código:

```
21.0
3.5
Traceback (most recent call last):
  File "C:/Users/Usuario/Dropbox/PYTHON/PROGRAMAS/4 zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/Users/Usuario/Dropbox/PYTHON/PROGRAMAS/4 zeroDivide.py", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

El error `ZeroDivisionError` ocurre siempre que intentamos dividir un número entre cero. Gracias al número de línea proporcionado en el mensaje de error, sabemos que es la sentencia `return` en la función `spam()` la que está causando el error.

Afortunadamente, podemos gestionar estos errores con las sentencias `try` y `except`. El código que potencialmente podría causar un error se pone dentro de una cláusula `try`. Si ocurre un error, la ejecución del programa salta al principio de la cláusula `except` relacionada con ese error.

Podemos poner el código previo dentro de una cláusula `try`, y añadir una cláusula `except` que contenga el código para gestionar lo que ocurre cuando se produce el error `ZeroDivisionError`:

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Cuando el código dentro de una cláusula `try` genera un error, la ejecución del programa salta inmediatamente al código en la cláusula `except`. Después de ejecutar ese código, la ejecución continúa normalmente.

Así, la salida del programa previo es la siguiente:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

Observar que también se detectará cualquier error que ocurra en las llamadas a una función que se hagan dentro de un bloque `try`. Consideremos el siguiente programa, el cual tiene las llamadas a la función `spam()` dentro del bloque `try`:

```
def spam(divideBy):
    return 42 / divideBy

try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

Cuando ejecutamos este programa, la salida que obtenemos es:

```
21.0
3.5
Error: Invalid argument.
```

La razón por la que la instrucción `print(spam(1))` no se ejecuta nunca es que, una vez que la ejecución salta al código dentro de la cláusula `except`, ya no retorna a la cláusula `try`. En vez de eso, el programa simplemente continúa ejecutándose hacia abajo normalmente.

4.9. EJERCICIOS DEL CAPÍTULO 4.

Ejercicio 4.1. Escribe una función llamada `oddEven()` a la que se le pase como argumento un número entero (llama a este parámetro `number`). La función debe determinar si ese número es par o impar. Tras la definición de la función, en el programa principal, haz unas cuantas llamadas a la función e imprime el resultado por pantalla, simplemente para comprobar si la función opera correctamente. Guarda el programa como `Ejer4.1.py`.

Ejercicio 4.2. Como sabemos, la función integrada `abs()` nos permite obtener el **valor absoluto** de un número cualquiera. Supón que Python no proporciona esa función; vamos a programarla nosotros. Para ello, escribe una función llamada `absValue()` que reciba un parámetro llamado `number`. Escribe el código de esta función para que funcione como se espera. Tras la definición de la función, en el programa principal, haz unas cuantas llamadas a la función e imprime el resultado por pantalla, simplemente para comprobar si la función opera correctamente. Guarda el programa como `Ejer4.2.py`.

Ejercicio 4.3. Escribe una función a la que se le pase un número del 1 al 7, y que devuelva el día de la semana correspondiente. (Por ejemplo, si le pasas un 1, el programa devuelve "lunes"; si le pasas un 2, el programa devuelve "martes"; y si le pasas un 7, el programa devuelve "domingo"). Haz que el programa responda adecuadamente si el usuario no introduce un número correcto. (Por ejemplo, si el usuario inserta un 0 ó un 8, el programa debe indicar que se ha equivocado, y que inserte un número del 1 al 7). Guarda el programa como `Ejer4.3.py`.

Ejercicio 4.4. El **factorial** de un número se define como:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

Sabemos que el módulo `math` incluye una función para calcular el factorial de un número, a saber, `math.factorial()`. Sin embargo, imagina que esa función no está disponible. El problema consiste en escribir la función que nos permita calcular el factorial de un número entero (por ejemplo, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$). Como sabrás, la operación factorial es muy importante en combinatoria y en el cálculo de probabilidades.

Al calcular el factorial, hay ciertos casos especiales que el programa debe poder manejar; en particular, se cumple que:

$$0! = 1$$

Resuelve este ejercicio de dos formas distintas, a saber, con un bucle `for` y con un bucle `while`. Guarda el programa como `Ejer4.4.py`.

Ejercicio 4.5. Números combinatorios.
Basándote en la función del ejercicio previo, escribe una función que calcule la cantidad:

$$\binom{N}{n} \equiv \frac{N!}{n!\,(N-n)!}$$

Esta cantidad se denomina **número combinatorio**, y se lee "N sobre n". Por supuesto, la función debe recibir como argumentos dos enteros N y n, con $N > n$, y devolver un entero que sea el número combinatorio correspondiente. Como ya sabrás, el número combinatorio es una cantidad de interés en combinatoria y probabilidad. Guarda el programa como `Ejer4.5.py`.

Ejercicio 4.6. Usando la función del ejercicio previo, escribe un programa que imprima las 20 primeras líneas del "triángulo de Pascal". La línea n −ésima del triángulo de Pascal consta de $n+1$ números, que son

los números combinatorios $\binom{n}{0}$, $\binom{n}{1}$, …, $\binom{n}{n}$. Por ejemplo, para obtener la primera línea deberías calcular $\binom{1}{0}$ y $\binom{1}{1}$; para la segunda línea $\binom{2}{0}$, $\binom{2}{1}$, y $\binom{2}{2}$; para la tercera $\binom{3}{0}$, $\binom{3}{1}$, $\binom{3}{2}$, y $\binom{3}{3}$, etc. Las 4 primeras líneas del triángulo de Pascal son:

```
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Ejercicio 4.7. La probabilidad de que al lanzar una moneda (no trucada) n veces obtengamos k caras viene dada por:

$$P = \frac{casos\ favorables}{casos\ posibles} = \frac{\binom{n}{k}}{2^n}$$

Basándote en la función del ejercicio 4.6, escribe una función que reciba como parámetros los valores n y k, y que obtenga la probabilidad buscada.
(a) ¿Cuál es la probabilidad de que, al lanzar la moneda 100 veces, obtengamos exactamente 60 caras?
(b) Escribe un programa para calcular la probabilidad de que, al lanzar una moneda 100 veces, obtengamos 60 o más caras.

Ejercicio 4.8. Escribe una función que reciba como parámetros las coordenadas Cartesianas x e y de un punto en el plano, y que devuelva en qué cuadrante se sitúa ese punto. Guarda el programa como `Ejer4.8.py`.

Ejercicio 4.9. Escribe una función que te permita hallar el número más grande de entre tres números. Por ejemplo, si a la función se le pasa los números $4, -5$, y 8, la función debería devolver 8.

Pista: Compara los números de dos en dos, sucesivamente. Esta técnica puede generalizarse para hallar el número más grande de entre un conjunto de números arbitrariamente grande. Guarda el programa como `Ejer4.9.py`.

Ejercicio 4.10. Escribe una función que, dados cuatro números, devuelva el mayor producto de dos de ellos. Por ejemplo, si la función recibe los números $1, 5, -2, -4$ debe devolver 8, que es el producto más grande que se puede obtener entre ellos. Guarda el programa como `Ejer4.9.py`.

Ejercicio 4.11. La secuencia de Collatz.
Escribe una función llamada `collatz()` que tenga un parámetro llamado `number`. Si `number` es par, la función `collatz()` debería imprimir `number//2` y devolver este valor. Si `number` es impar, la función `collatz()` debería imprimir y devolver `3*number+1`.

A continuación, escribe un programa que le permita al usuario escribir un entero y que llame continuamente a la función `collatz()` sobre ese número hasta que la función devuelva el valor 1. (Sorprendentemente, esta secuencia funciona para cualquier entero; usando esta secuencia, antes o después terminaremos llegando al valor 1. Incluso los matemáticos no tienen claro el por qué. Este programa explora la denominada **secuencia de Collatz**, a veces llamada "el problema imposible más fácil de las matemáticas"). Acuérdate de convertir el valor de retorno de la función `input()` a un entero con la función `int()`; en caso contrario, el programa estará usando un valor de tipo cadena. (PISTA: Un número entero es par si el resto de su división entre dos es cero, e impar si ese resto es 1).

La salida de este programa debería parecerse a lo siguiente:

```
Enter number:
3
10
5
16
8
4
2
1
```

Añade unas sentencias `try` y `except` al proyecto previo para detectar si el usuario escribe una cadena que no sea un número entero. Normalmente, la función `int()` generará un error `ValueError` si se le pasa una cadena no entera, como por ejemplo `int('dog')`. En la cláusula del `except`, imprime un mensaje para el usuario que le diga que debe introducir un entero. Guarda el programa como `Ejer4.11.py`.

Ejercicio 4.12. Una característica muy útil de las funciones es la **recursión**, esto es, la capacidad de una función de llamarse a sí misma.
(a) Considera la siguiente definición alternativa del factorial $n!$ de un entero positivo n:

$$n! = \begin{cases} 1 & si\ n = 1 \\ n \times (n-1)! & si\ n > 1 \end{cases}$$

Escribe una función que calcule el factorial de un entero n aplicando esta definición.
(b) En el capítulo 3 ya nos encontramos con los números de Catalan. Como vimos, estos números se definen de forma recursiva como:

$$C_n = \begin{cases} 1 & si\ n = 0 \\ \frac{4n-2}{n+1} C_{n-1} & si\ n > 0 \end{cases}$$

Escribe una función en Python que calcule C_n usando recursión. Usa tu función para calcular e imprimir C_{100}.
(c) Euclides demostró que el máximo común divisor `gcd(m,n)` de dos enteros no negativos m y n satisface:

$$gcd(m,n) = \begin{cases} m & si\ n = 0 \\ gcd(n, m \bmod n) & si\ n > 0 \end{cases}$$

Escribe una función en Python, `gcd(m,n)`, que use recursión para obtener el máximo común divisor de dos números m y n con esta fórmula. Usa tu función para calcular e imprimir el máximo común divisor de unas cuentas parejas de enteros.

Ejercicio 4.13. Coste de un viaje.
Vamos a usar funciones para calcular el coste de un viaje. Para ello:

- Define una función llamada `hotel_cost` con dos argumentos llamados `nights` (un entero) y `category` (una cadena) como entradas. Si la categoría del hotel es básica ('basic') el precio por noche es de 90€. Si la categoría del hotel es media ('medium') el precio por noche es 110€. Y si la categoría del hotel es alta ('premium'), el precio por noche es 160€.
- Define una función llamada `flight_cost` que reciba dos argumentos, `city` y `luggage` como entradas. `city` será una cadena que especifique el destino, y `luggage` un entero que indique el número de maletas facturadas. La función debe devolver un precio diferente dependiendo de la

ciudad de destino. Los destinos válidos y sus precios correspondientes son: Paris: 120€, Rome: 95€, London: 145€, Stockholm: 225€. Además, por cada maleta facturada, el precio subirá en 35€.

- Define una función llamada `car_rental_cost` con dos argumentos llamados `car_type` y `days`, que calcule el coste de alquiler de un coche. `Type` es una cadena que será `'manual'` o `'automatic'`, y `days` será el número de días de alquiler. Cada día que alquilemos el coche cuesta 40€, pero si lo alquilamos durante 7 días o más, obtenemos un descuento en el precio equivalente a un día de alquiler. Alternativamente, si alquilamos el coche durante tres días o más, obtenemos un descuento de 20€. No podemos obtener ambos descuentos simultáneamente. Por otra parte, si optamos por un coche automático, el precio del alquiler sube a 50€ por día (con los descuentos idénticos al caso del coche manual).
- Por último, define una función llamada `trip_cost` que reciba los argumentos, `city`, `days`, `luggage`, `hotel_category`, y `car_type`. Esta función debe calcular el coste total del viaje.

Ejercicio 4.14. Piedra, papel, o tijeras.
En este ejercicio te proponemos programar el juego de piedra, papel o tijeras (rock, paper, scissors) para dos jugadores. Recuerda las reglas:

- La piedra vence a las tijeras (porque las machaca).
- Las tijeras vencen al papel (porque lo corta).
- El papel vence a la piedra (porque la envuelve).

Pista 1: Pide a cada jugador su apuesta, compara sus respuestas, imprime un mensaje que indique qué jugador es el ganador, y pregunta a los jugadores si quieren comenzar una nueva partida (y actúa en consecuencia).
Pista 2: Si dos jugadores empatan (digamos, dos tijeras) no hay ganador, así que el programa les debe pedir que vuelvan a elegir hasta que deje de haber empate.
Pista 3: Te conviene crear funciones para que el programa sea más fácil de escribir y de leer.

Ejercicio 4.15. Cuando se excitan mediante descargas eléctricas, los átomos emiten luz únicamente en unas longitudes de onda (esto es, en ciertos colores) que son características del elemento implicado. La técnica de la espectroscopia permitió medir de forma precisa esas longitudes de onda. Sin embargo, la física teórica se mostró incapaz de explicar la razón por la que esto ocurría, hasta que Niels Bohr propuso su **teoría del átomo de hidrógeno**:

Según Bohr, los electrones del átomo no pueden poseer cantidades arbitrarias de energía, sino que únicamente pueden existir únicamente en ciertos **estados estacionarios** cuya energía viene dada por:

$$E_n = -\frac{m_e e^4}{8\varepsilon_0^2 h^2}\frac{1}{n^2}$$

, donde $m_e = 9{,}1094 \times 10^{-31}\, kg$ es la masa del electrón, $e = 1{,}6022 \times 10^{-19}\, C$ es la carga elemental, $\varepsilon_0 = 8{,}8542 \times 10^{-12}\ C^2 s^2/kg\, m^3$ es la permitividad eléctrica del vacío, $h = 6{,}6261 \times 10^{-34}\, J\, s$ es la constante de Planck, y n identifica el nivel de energía en el que se encuentra el electrón ($n = 1,\ 2,\ 3,\ 4$ ...).

Mientras el electrón se encuentra en un estado estacionario E_n, el átomo no radia luz. Sin embargo, un electrón puede saltar desde un nivel E_n a un nivel E_m de menor energía, lo que implica la emisión de luz con una energía igual a la diferencia de energías entre los niveles:

$$\Delta E = E_n - E_m = -\frac{m_e e^4}{8\varepsilon_0^2 h^2}\left(\frac{1}{n^2} - \frac{1}{m^2}\right) = \frac{m_e e^4}{8\varepsilon_0^2 h^2}\left(\frac{1}{m^2} - \frac{1}{n^2}\right)$$

Previamente a los trabajos de Bohr, Einstein había postulado que la energía transmitida por la luz consistía en "paquetes" discretos, llamados **fotones**. Según Einstein, cada fotón transporta una energía $E = hf$, donde f es la frecuencia de la luz. Empleando la relación $c = \lambda f$ (donde c es la rapidez de la luz, y λ la

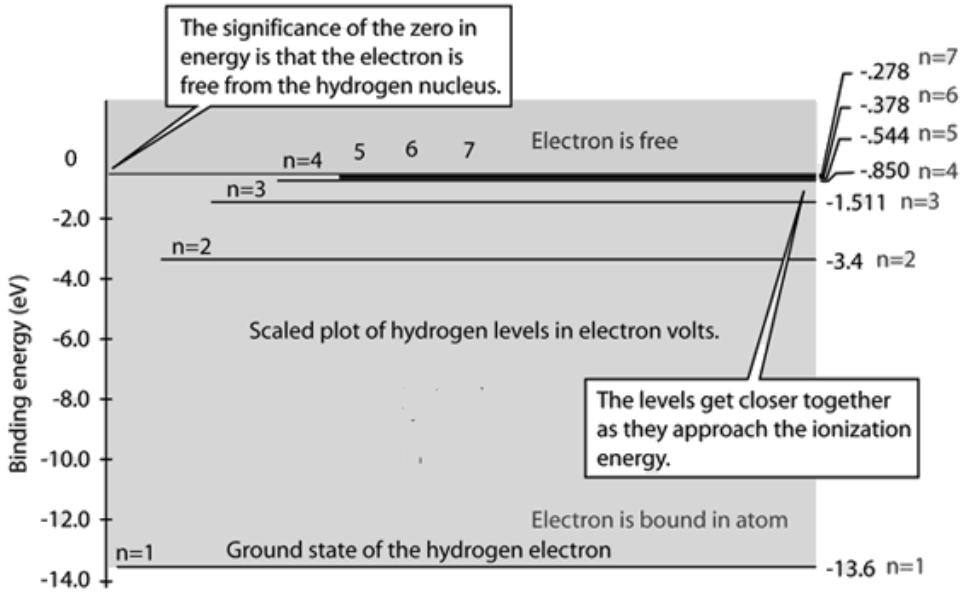
longitud de onda), tenemos que la longitud de onda de la luz emitida por el átomo de hidrógeno cuando un electrón pasa del nivel n al nivel m puede obtenerse como:

$$E = E_n - E_m = hf = \frac{hc}{\lambda}$$

, esto es:

$$\frac{1}{\lambda} = \frac{(E_n - E_m)}{hc} = \frac{m_e e^4}{8\varepsilon_0^2 h^3 c}\left(\frac{1}{m^2} - \frac{1}{n^2}\right)$$

(a) Escribe una función que calcule el nivel de energía E_n de un átomo de hidrógeno, recibiendo como argumento el valor de n (donde $n = 1,\ 2,\ 3,\ 4$ ...). La fórmula te dará la energía en julios (J). Convierte a unidades de electrón – voltios (eV) aplicando la conversión $1\ eV = 1{,}6022 \times 10^{-19}\ J$, y busca en internet "niveles de energía del átomo de hidrógeno" para comprobar tus resultados. CUIDADO: en este ejercicio es muy importante poner paréntesis para que las fórmulas que programes funcionen correctamente. Guarda el programa como `Ejer4.15a.py`.

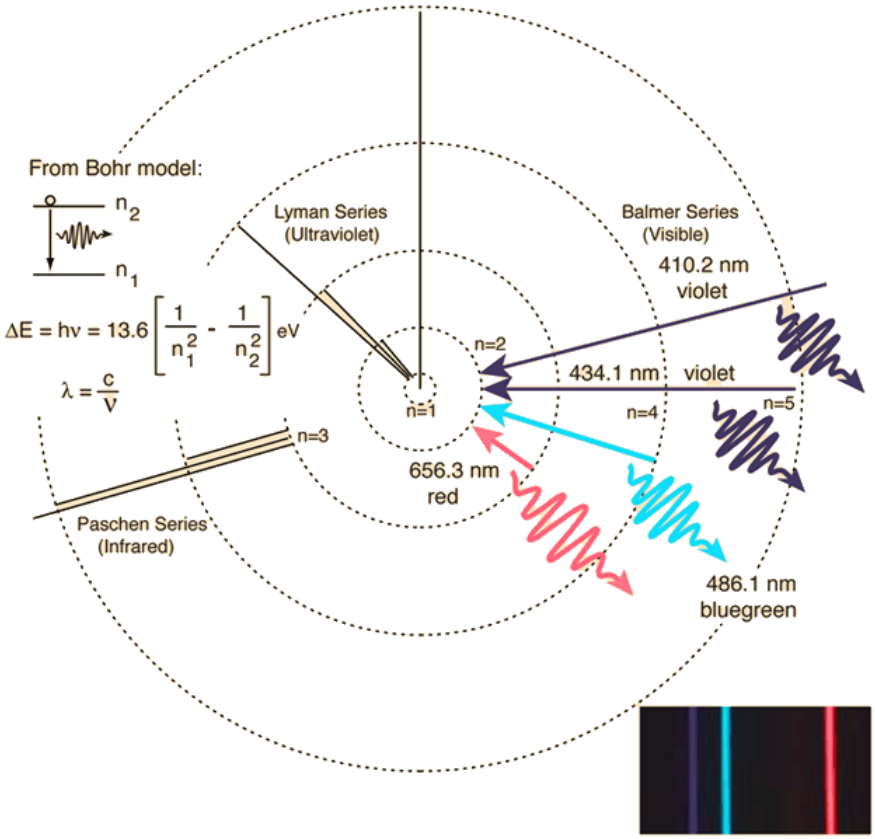


(b) Escribe un programa que calcule las longitudes de onda de la luz emitida por el átomo de hidrógeno (el llamado **espectro de emisión del hidrógeno**), basándote en la fórmula:

$$\frac{1}{\lambda} = \frac{E_n}{hc} - \frac{E_m}{hc}$$

Pista 1: necesitarás dos bucles anidados: El bucle externo recorrerá los niveles $m = 1, 2, 3$, y para cada uno de estos valores, un bucle interno calculará la longitud de onda de la luz emitida debido a la transición desde los niveles $n = m + k$ (con $k = 1,\ 2,\ 3,\ 4,\ 5$) a ese nivel m.
Pista 2: Utiliza la función que te permitía hallar la energía E_n de un nivel cualquiera.

Esta fórmula te da la longitud de onda en metros. Convierte de metros (m) a angstroms (Å) aplicando la conversión $1\ \text{Å} = 10^{-10} m$, y muestra las longitudes de onda tanto en metros como en angstroms. Busca en internet "espectro de emisión del hidrógeno" para comprobar tus resultados. Guarda el programa como `Ejer4.15b.py`.



Ejercicio 4.16. Combinatoria.

La **combinatoria** es la rama de las matemáticas que estudia las diversas formas de realizar agrupaciones con los elementos de un conjunto, y de contar el número de agrupaciones posibles. Existen diversas formas de realizar agrupaciones, dependiendo de si se pueden repetir o no los elementos, de si se pueden tomar todos los elementos de los que disponemos, y de si influye o no en orden de colocación de esos elementos. En función de todo ello, podemos distinguir tres tipos de agrupaciones: **variaciones** (con y sin repetición), **permutaciones** (con y sin repetición), o **combinaciones**. Suponer que tenemos N elementos disponibles, y que tomamos n de esos elementos.

> NOTA: Para un repaso de la combinatoria, puedes visitar la página web de vitutor, https://www.vitutor.com/pro/1/analisis_combinatorio.html, o releer el tema de combinatoria en tu libro de matemáticas de 4º ESO.

Para poder determinar el tipo de agrupación que tenemos en un problema en particular, hemos de responder a estas preguntas:

¿Importa el orden de colocación de los elementos?
Sí ⇒ variaciones o permutaciones.
No ⇒ combinaciones.

¿Tomamos todos los elementos disponibles, o solo algunos?
Cogemos todos ⇒ permutaciones $(N = n)$.
Tomamos solo algunos ⇒ variaciones.

¿Se pueden repetir los elementos?
Sí ⇒ variaciones o permutaciones con repetición.
No ⇒ variaciones o permutaciones sin repetición.

La tabla resume todas las posibilidades:

		AGRUPACIONES	Sin repetición	Con repetición
¿Importa el orden de colocación?	**SI**	**Variaciones**. (Tomamos algunos elementos: $n < N$)	$V_N^n = \frac{N!}{(N-n)!} =$ $= N \times (N-1) \times \ldots \times (N-n+1)$	$VR_N^n = N^n$
		Permutaciones. (Tomamos todos los elementos: $n = N$)	$V_N^N = P_n = n!$	$P_n = \frac{n!}{a!\,b!\,c!\,\ldots}$ $a + b + c + \cdots = n$
	NO	**Combinaciones**.	$C_N^n = \frac{Variaciones}{Permutaciones} = \frac{V_N^n}{P_n} = \frac{N!}{n!\,(N-n)!} \equiv \binom{N}{n}$	

, donde $a, b, c, \ldots$ son el número de veces que se repite el $1º, 2º, 3º, \ldots$ elemento.

Crea un programa que empiece pidiendo el número de elementos disponibles, N, y el número de elementos que tomamos, n. A continuación, el programa te va haciendo preguntas. ¿Importa el orden de colocación de los elementos? ¿Tomamos todos los elementos disponibles? ¿Se pueden repetir los elementos?, y en función de la respuesta, aplica la fórmula adecuada y responde con el número de agrupaciones posibles. Guarda el programa como `Ejer4.15.py`.

PISTA 1: Ve almacenando las respuestas (`Y` or `N`) en varias variables, por ejemplo, `orderMatters`, `allElements`, y `repAllowed`, y mapea todas las combinaciones posibles. Por ejemplo, si importa el orden de colocación, tomamos todos los elementos, y no hay repetición, tenemos permutaciones sin repetición).

PISTA 2: Utiliza llamadas a la función que escribiste en el ejercicio 4.3 para calcular todos los factoriales de las distintas fórmulas.

PISTA 3: Las permutaciones con repetición tendrás que tratarlas con cuidado: Dado el número total n de elementos (los cuales pueden repetirse), deberás preguntar cuántas veces (a) se repite el primer elemento, cuántas veces (b) se repite el segundo elemento, cuántas veces (c) se repite el segundo elemento, etc., hasta que se cumpla que $a + b + c + \cdots = n$.

5. LISTAS.

Un concepto más que debemos conocer antes de ponernos a escribir programas en serio es el tipo de datos **lista**, así como su pariente, el tipo de datos **tupla**. Las listas y las tuplas pueden contener múltiples valores, lo que nos facilita enormemente escribir programas que tengan que manejar grandes cantidades de datos. Y como las propias listas pueden contener otras listas, podemos usarlas para organizar los datos en estructuras jerárquicas.

En este capítulo discutiremos los fundamentos de las listas. También estudiaremos los **métodos**, que son funciones que están vinculadas a valores de un cierto tipo de datos. Por último, trataremos con las tuplas (en su versión más parecida a las listas), hablaremos del tipo de datos cadena, y discutiremos en qué se diferencian de las listas. En el próximo capítulo presentaremos el tipo de datos **diccionario**.

5.1. EL TIPO DE DATOS LISTA.

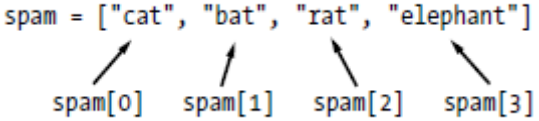
Una **lista** es un valor que contiene múltiples valores en una secuencia ordenada. El término *valor tipo lista* se refiere a la propia lista (un valor que puede almacenarse en una variable o pasarse como argumento a una función como otro valor cualquiera), y no a los valores que hay dentro de ella. Un valor tipo lista puede parecerse a lo siguiente: `['cat','bat','rat','elephant']`. De la misma forma que los valores tipo cadena se escriben entre comillas simples para indicar dónde empieza y termina la cadena, las listas empiezan abriendo con un corchete y terminan cerrando con otro corchete, `[]`. Los valores dentro de la lista se denominan **objetos** (items). Los objetos están separados por comas. A modo de ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 'elephant']
['hello', 3.1415, True, None, 'elephant']
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

Notar que a la variable `spam` solo se le asigna un único valor: el valor tipo lista. Pero la propia lista contiene otros valores. El valor `[]` es una lista vacía que no contiene valores, similar a la cadena vacía `''`.

OBTENER LOS VALORES INDIVIDUALES DE UNA LISTA CON ÍNDICES.

Suponer que tenemos la lista `['cat','bat','rat','elephant']` almacenada en la variable `spam`. El código Python `spam[0]` se evaluaría a `'cat'`, el código `spam[1]` se evaluaría a `'bat'`, y así sucesivamente. El entero dentro de los corchetes que viene tras el nombre de la lista se denomina **índice**. El primer valor de la lista está en el índice `0`, el segundo valor está en el índice `1`, el tercer valor en el índice `2`, etc. La figura muestra un valor tipo lista asignado a la variable `spam`, junto con la forma en la que se evaluarían las expresiones con índices.



A modo de ejemplo, escribe las siguientes expresiones en el shell interactivo. Comienza asignando una lista a la variable `spam`:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
❶ >>> 'Hello '+spam[0]
❷ 'Hello cat'
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
'The bat ate the cat.'
```

Notar que la expresión `'Hello '+spam[0]` se evalúa a `'Hello ' + 'cat'`, porque `spam[0]` se evalúa a la cadena `'cat'`. A su vez, esta expresión se evalúa a la cadena `'Hello cat'`.

Python nos dará un error de tipo `IndexError` si usamos un índice que exceda el número de valores en nuestra lista:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1000]
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    spam[1000]
IndexError: list index out of range
```

Los índices solo pueden ser valores enteros, no flotantes. El siguiente ejemplo causa un error de tipo `TypeError`:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
'bat'
>>> spam[1.0]
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    spam[1.0]
TypeError: list indices must be integers or slices, not float
>>> spam[int(1.0)]
'bat'
```

Las listas también pueden contener otros valores de tipo lista. Podemos acceder a los valores dentro de estas listas usando índices múltiples, como por ejemplo:

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

El primer índice indica qué valor tipo lista usar, y el segundo indica el valor dentro de ese valor tipo lista. Por ejemplo, `spam[0][1]` imprime `'bat'`, esto es, el segundo valor dentro de la primera lista. Si solo usamos un índice, el programa imprimirá toda la lista situada en ese índice.

ÍNDICES NEGATIVOS.

Aunque los índices empiezan con el 0 y van aumentando, también podemos usar enteros negativos. El valor -1 se refiere al último índice de la lista, el valor -2 al penúltimo índice de la lista, y así sucesivamente. Escribe lo siguiente en el shell interactivo:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
'The elephant is afraid of the bat.'
```

OBTENER SUBLISTAS MEDIANTE CORTES.

De la misma forma que un índice nos permite obtener un valor de una lista, un **corte** (slice) nos permite obtener varios valores de una lista. Los cortes se escriben entre corchetes, igual que los índices, pero tienen dos enteros separados por dos puntos. Observa la diferencia entre los índices y los cortes:

- `spam[2]` es una lista con un índice (un entero).
- `spam[1:4]` es una lista con un corte (dos enteros separados por dos puntos).

En un corte, el primer entero es el índice donde empieza el corte, y el segundo entero es el índice donde termina el corte. Un corte llegará hasta, pero no incluirá, el valor del segundo índice. Un corte se evalúa a un nuevo valor tipo lista. Escribe lo siguiente en el shell interactivo:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

A modo de atajo, podemos dejar de poner uno o los dos índices de un corte. Si no ponemos el primer entero, es lo mismo que poner un 0, esto es, el principio de la lista. Si no ponemos el segundo índice es lo mismo que usar la longitud de la lista, lo que hará que el corte llegue hasta el final de la lista. Escribe lo siguiente en el shell interactivo:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

OBTENER LA LONGITUD DE UNA LISTA CON LEN().

La función `len()` devolverá el número de valores que hay en el valor tipo lista que le pasemos como argumento, de la misma forma que cuenta el número de caracteres de un valor tipo cadena.

Inserta lo siguiente en el shell interactivo:

```
>>> spam = ['cat', 'dog', 'moose', [10, 20, 30, 40, 50]]
>>> len(spam)
4
```

CAMBIAR LOS VALORES DE UNA LISTA CON ÍNDICES.

Normalmente, el nombre de una variable va a la izquierda de una sentencia de asignación, como por ejemplo, `spam = 42`. Sin embargo, también podemos usar un índice de una lista para cambiar el valor en ese índice. Por ejemplo, `spam[1] = 'aardvark'` significa "asigna al valor situado en el índice 1 de la lista `spam` la cadena `'aardvark'`". Escribe lo siguiente en el shell interactivo:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1] = 'aardvark'
>>> spam
['cat', 'aardvark', 'rat', 'elephant']
>>> spam[2] = spam[1]
>>> spam
['cat', 'aardvark', 'aardvark', 'elephant']
>>> spam[-1]=12345
>>> spam
['cat', 'aardvark', 'aardvark', 12345]
```

CONCATENACIÓN Y RÉPLICA DE LISTAS.

El operador `+` puede combinar dos listas para crear un nuevo valor de tipo lista de la misma forma que combina dos cadenas para crear un nuevo valor de tipo cadena. También podemos usar el operador `*` con una lista y un entero para replicar la lista tantas veces como indique el entero. Escribe lo siguiente en el shell interactivo:

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

QUITAR VALORES DE UNA LISTA CON LA SENTENCIA DEL.

La sentencia `del` borra los valores localizados en el índice que se le pasa como argumento. Todos los valores de la lista situados detrás del valor borrado se mueven un índice hacia abajo. Por ejemplo, inserta lo siguiente en el shell interactivo:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat']
```

También podemos usar la sentencia `del` sobre una variable para borrarla, como si de una sentencia de "des-asignación" se tratara. Si intentamos usar la variable después de borrarla, obtendremos un error de tipo `NameError`, porque la variable ya no existe.

En la práctica, casi nunca tendremos que borrar variables. La sentencia `del` se usa principalmente para borrar valores de una lista.

5.2. TRABAJANDO CON LISTAS.

Al empezar a programar, resulta muy tentador crear muchas variables individuales para almacenar un grupo de valores similares. Por ejemplo, si quisiéramos almacenar los nombres de una camada de gatos, podría ser tentador escribir un código como éste:

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

Pero resulta que esta forma de escribir código no es la más adecuada. En efecto, si el número de gatos cambia, nuestro programa no será capaz de almacenar más gatos que variables tiene el programa. Además, este tipo de programas también suelen tener un montón de código duplicado o prácticamente idéntico. Por ejemplo, consideremos la cantidad de código duplicado que hay en el siguiente programa. Escríbelo en el editor de archivos y guárdalo con el nombre `allMyCats1.py`.

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
catName2 = input()
print('Enter the name of cat 3:')
catName3 = input()
print('Enter the name of cat 4:')
catName4 = input()
print('Enter the name of cat 5:')
catName5 = input()
print('Enter the name of cat 6:')
catName6 = input()
print('The cat names are:')
print(catName1+' '+catName2+' '+catName3+' '
      +catName4+' '+catName5+' '+catName6)
```

En vez de usar múltiples variables repetitivas, podemos usar una sola variable que contenga un valor tipo lista. Por ejemplo, he aquí una nueva versión mejorada del programa `allMyCats1.py`. Esta nueva versión usa una lista y puede almacenar tantos gatos como el usuario quiera introducir. Escribe el siguiente código en el editor de archivos y guárdalo como `allMyCats2.py`:

```
catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) +
          ' (Or enter nothing to stop):')
    name = input()
    if name == '':
        break
    catNames = catNames + [name]  # list concatenation
print('The cat names are:')
for name in catNames:
    print('  ' + name)
```

Cuando ejecutemos este programa, la salida se parecerá a lo siguiente:

```
Enter the name of cat 1 (Or enter nothing to stop):
Zophie
Enter the name of cat 2 (Or enter nothing to stop):
Pooka
Enter the name of cat 3 (Or enter nothing to stop):
Simon
Enter the name of cat 4 (Or enter nothing to stop):
Lady Macbeth
Enter the name of cat 5 (Or enter nothing to stop):
Fat-tail
Enter the name of cat 6 (Or enter nothing to stop):
Miss Cleo
Enter the name of cat 7 (Or enter nothing to stop):

The cat names are:
  Zophie
  Pooka
  Simon
  Lady Macbeth
  Fat-tail
  Miss Cleo
```

La ventaja de usar una lista es que nuestros datos ahora están en una estructura, por lo que nuestro programa es mucho más flexible para procesar los datos de lo que lo sería si los datos estuviesen en varias variables repetitivas. Por cierto, puede que no hayas entendido del todo la forma en la que hemos usado el bucle `for` en este ejemplo. A continuación, explicaremos cómo usar bucles `for` para recorrer los distintos elementos de una lista.

USAR BUCLES FOR CON LISTAS.

En el capítulo 3 aprendimos a usar bucles `for` para ejecutar un bloque de código un cierto número conocido de veces. Técnicamente, un bucle `for` repite el bloque de código una vez por cada valor que haya en una lista (o en un valor de tipo similar a una lista). Por ejemplo, si ejecutamos este código:

```
for i in range(4):
    print(i)
```

, la salida de este programa es:

```
0
1
2
3
```

Esto es así porque el valor de retorno de `range(4)` es un valor de tipo similar a una lista que Python trata con si se tratase de la lista `[0, 1, 2, 3]`. De hecho, el siguiente programa produce la misma salida que el anterior:

```
numList = [0, 1, 2, 3]
for number in numList:
    print(number)
```

El bucle `for` de este último programa itera a través de su cláusula con la variable `number` tomando los distintos valores de la lista `numList` sucesivamente en cada iteración.

NOTA: En este texto usaremos la denominación "valor de tipo similar a una lista" para referirnos a tipos de datos que técnicamente se denominan de forma común como **secuencias**.

Una técnica muy común es usar `range(len(someList))` con un bucle `for` para iterar sobre los índices de una lista. Por ejemplo, escribe el siguiente código en el shell interactivo:

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
>>> for i in range(len(supplies)):
        print('Index ' + str(i) + ' in supplies is: ' + supplies[i])

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

Haber usado `range(len(supplies))` en el bucle `for` previo ha sido muy útil, porque de esta forma el código dentro del bucle puede acceder al índice (con la variable `i`) y al valor en ese índice (con `supplies[i]`). Y lo mejor de todo, `range(len(supplies))` iterará a través de *todos* los índices de `supplies`, independientemente de cuántos objetos contenga esta lista.

LOS OPERADORES IN Y NOT IN.

Podemos saber si un cierto valor está o no está dentro de una lista con los operadores `in` y `not in`. Como cualquier otro operador, los operadores `in` y `not in` se usan en expresiones y conectan dos valores: el valor a buscar en la lista y la lista donde buscarlo. Estas expresiones se evaluarán a un valor Booleano. Escribe el siguiente código en el shell interactivo:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'hi-ya', 'hey']
True
>>> spam = ['hello', 'hi', 'howdy', 'hi-ya', 'hey']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

Por ejemplo, el siguiente programa le permite al usuario escribir un nombre de mascota y comprobar si ese nombre está en la lista de mascotas. Abre un nuevo editor de archivos, escribe el siguiente código, y guárdalo como `myPets.py`:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in myPets:
    print('I do not have a pet named ' + name)
else:
    print(name + ' is my pet.')
```

La salida del programa debería parecerse a lo siguiente:

```
Enter a pet name:
Garfield
I do not have a pet named Garfield
>>>
```

EL TRUCO DE LAS ASIGNACIONES MÚLTIPLES.

El **truco de las asignaciones múltiples** es un atajo que nos permite asignar a múltiples variables los valores almacenados en una lista usando una sola línea de código. Por lo tanto, en vez de hacer esto:

```
>>> cat = ['fat', 'black', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

, podríamos escribirlo todo en una sola línea de código, como:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition = cat
```

El número de variables y la longitud de la lista debe ser exactamente el mismo. En caso contrario, Python dará un error de tipo `ValueError`:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: not enough values to unpack (expected 4, got 3)
```

5.3. OPERADORES DE ASIGNACIÓN AUMENTADOS.

Al asignar un valor a una variable, es muy habitual usar la propia variable. Por ejemplo, después de asignarle el valor `42` a la variable `spam`, podríamos querer incrementar el valor de `spam` en `1` mediante el siguiente código:

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

A modo de atajo, podemos usar el operador aumentado `+=` para hacer exactamente lo mismo:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

En Python tenemos **operadores de asignación aumentados** para los operadores `+`, `-`, `*`, `/`, y `%`, tal y como se describen en la siguiente tabla:

Augmented assignment statement	Equivalent assignment statement
`spam = spam + 1`	`spam += 1`
`spam = spam - 1`	`spam -= 1`
`spam = spam * 1`	`spam *= 1`
`spam = spam / 1`	`spam /= 1`
`spam = spam % 1`	`spam %= 1`

El operador += también sirve para hacer concatenaciones de cadenas y concatenaciones de listas, y el operador *= puede hacer réplicas de cadenas y de listas. Inserta lo siguiente en el shell interactivo:

```
>>> spam = 'Hello'
>>> spam += ' world!'
>>> spam
'Hello world!'
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

5.4. MÉTODOS.

Un **método** es lo mismo que una función, excepto que a los métodos se les llama "sobre un valor" de un tipo determinado. Por ejemplo, si la variable `spam` almacenase un valor tipo lista, podríamos llamar al método `index()` (del que hablaremos más adelante) sobre esa lista de la siguiente forma: `spam.index('hello')`. Notar que el nombre del método viene después del nombre de la variable que almacena el valor, separado por un punto.

Cada tipo de datos tiene su propio conjunto de métodos. En particular, el tipo de datos lista tiene varios métodos útiles para encontrar, añadir, quitar, y manipular de otras muchas formas los valores de una lista.

ENCONTRAR UN VALOR EN UNA LISTA CON EL MÉTODO INDEX().

Los valores tipo lista tienen un método `index()` al que se le puede pasar un valor, y si ese valor existe en la lista, devuelve el índice de ese valor. Si el valor no está en la lista, Python produce un error `ValueError`. Escribe lo siguiente en el shell interactivo:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
>>> spam.index('hiya')
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    spam.index('hiya')
ValueError: 'hiya' is not in list
```

Cuando hay valores duplicados en la lista, `index()` devuelve el índice de su primera aparición. Escribe lo siguiente en el shell interactivo, y observa que `index()` devuelve `1`, y no `3`.

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
1
```

AÑADIR VALORES A UNA LISTA CON LOS MÉTODOS APPEND() E INSERT().

Para añadir nuevos valores a una lista, usamos los métodos `append()` e `insert()`. Escribe lo siguiente en el shell interactivo para llamar al método `append()` sobre un valor tipo lista que está almacenado en la variable `spam`:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
>>> spam
['cat', 'dog', 'bat', 'moose']
```

La llamada al método `append()` añade el argumento al final de la lista. El método `insert()` nos permite insertar un valor en un índice cualquier de la lista. El primer argumento de `insert()` es el índice del nuevo valor, y el segundo argumento es el nuevo valor a insertar. Escribe lo siguiente en el shell interactivo:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

Notar que escribimos `spam.append('moose')` y `spam.insert(1, 'chicken')`, y no `spam = spam.append('moose')` ni `spam = spam.insert(1, 'chicken')`. Ni `append()` ni `insert()` le dan un nuevo valor a `spam` como su valor de retorno. (De hecho, el valor de retorno de `append()` e `insert()` es `None`, por lo que es evidente que no querremos guardar este valor como el nuevo valor de la variable). Hablaremos más de todas estas cosas en la sección "Tipos de datos mutables e inmutables".

Los métodos pertenecen a un solo tipo de datos. Por ejemplo, los métodos `append()` e `insert()` son métodos de listas, y solo pueden llamarse sobre valores tipo lista, y no sobre otros valores como enteros o cadenas. Escribe lo siguiente en el shell interactivo, y observa los mensajes de error `AttributeError` que aparecen.

```
>>> eggs = 'hello'
>>> eggs.append('world')
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    eggs.append('world')
AttributeError: 'str' object has no attribute 'append'
>>> bacon = 42
>>> bacon.insert(1, 'world')
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    bacon.insert(1, 'world')
AttributeError: 'int' object has no attribute 'insert'
```

QUITAR VALORES DE UNA LISTA CON REMOVE().

Al método `remove()` se le pasa el valor a quitar de la lista sobre la que se le llama. Escribe lo siguiente en el shell interactivo:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

Si intentamos borrar un valor que no existe en la lista, Python devolverá un error `ValueError`. Por ejemplo, escribe lo siguiente en el shell interactivo, y observa el error que aparece por pantalla:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

Si el valor aparece varias veces en la lista, solo se borrará la primera instancia. Por ejemplo:

```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
>>> spam.remove('cat')
>>> spam
['bat', 'rat', 'cat', 'hat', 'cat']
```

La sentencia `del` es útil cuando sabemos el índice del valor que queremos eliminar de la lista. Por su parte, el método `remove()` es útil cuando conocemos el valor que queremos quitar de la lista.

CLASIFICAR LOS VALORES DE UNA LISTA CON EL MÉTODO SORT().

Las listas de valores numéricos o de valores tipo cadena pueden clasificarse mediante el método `sort()`. Por ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> spam =  [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

También podemos pasar el valor `True` al argumento clave `reverse` para hacer que `sort()` clasifique los valores en orden inverso:

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

Hay tres cosas que debemos de tener en cuenta acerca del método `sort()`:

(1) Primero, el método `sort()` clasifica las listas "in situ"; no debemos intentar capturar el valor de retorno escribiendo un código como `spam = spam.sort()`.

(2) En segundo lugar, no podemos ordenar listas que contengan tanto números como cadenas, ya que Python no sabe cómo comparar valores de distinto tipo. Escribe lo siguiente en el shell interactivo y observa el mensaje de error `TypeError`:

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    spam.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

(3) Por último, el método `sort()` utiliza el "orden ASCIIbético" para ordenar cadenas, en vez de usar el orden alfabético real. Esto significa que las letras mayúsculas vienen antes que las letras minúsculas. Por consiguiente, la letra minúscula `a` se clasificará detrás que la letra mayúscula `Z`. He aquí un ejemplo:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

Si necesitamos clasificar los valores en orden alfabético normal, debemos pasar el valor `str.lower` al argumento clave `key` en la llamada al método `sort()`:

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

Esto hace que la función `sort()` trate todos los objetos de la lista como si estuviesen escritos en minúscula, pero sin cambiar los valores de la lista.

5.5. PROGRAMA DE EJEMPLO: BOLA MÁGICA CON LISTAS.

Usando listas podemos escribir una versión mucho más elegante del programa de la bola mágica que escribimos en la sección 4.3 del capítulo previo. En vez de tener varias sentencias `elif` casi idénticas, podemos crear una única lista con la que trabajará el programa. Abre un nuevo editor de archivos, escribe el siguiente código, y guárdalo como `magic8Ball2.py`:

```
import random

messages = ['It is certain',
    'It is decidedly so',
    'Yes definitely',
    'Reply hazy try again',
    'Ask again later',
    'Concentrate and ask again',
    'My reply is no',
    'Outlook not so good',
    'Very doubtful']

print(messages[random.randint(0, len(messages) - 1)])
```

Al ejecutar este programa, observaremos que funciona exactamente igual que el programa `magic8Ball.py` previo.

Notar la expresión usada como índice en `messages[random.randint(0, len(messages) - 1)]`. Esto produce un número aleatorio para usar como índice, independientemente del tamaño de la lista `messages`. Esta expresión permite obtener un valor aleatorio entre `0` y el valor de `len(messages) - 1`. La ventaja de este nuevo enfoque es que podemos añadir y quitar cadenas de la lista `messages` fácilmente, sin tener que cambiar otras líneas de código. Si en el futuro queremos actualizar nuestro código, habrá menos líneas de código que tendremos que cambiar y menos probabilidades de producir malfuncionamientos.

5.6. TIPOS SIMILARES A LISTAS: CADENAS Y TUPLAS.

Las listas no son los únicos tipos que nos permiten representar secuencias ordenadas de valores. Por ejemplo, las cadenas y las listas son, en realidad, muy similares, si consideramos que una cadena es una "lista" de caracteres individuales.

Muchas de las cosas que podemos hacer con las listas también las podemos hacer con las cadenas: indexado, troceado, uso de bucles `for`, de la función `len()`, y de los operadores `in` y `not in`.

Para entender todo esto, escribe lo siguiente en el shell interactivo:

```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
        print('* * * ' + i + ' * * *')


* * * Z * * *
* * * o * * *
* * * p * * *
* * * h * * *
* * * i * * *
* * * e * * *
```

Excepciones a las reglas de tabulación en Python.

En la mayoría de los casos, la cantidad de tabulación de una línea de código le dice a Python a qué bloque de código pertenece esa línea. Sin embargo, hay algunas excepciones a esta regla. Por ejemplo, las listas pueden extenderse varias líneas en un programa. La tabulación de esas líneas no es relevante; Python sabe que hasta que no vea el corchete de cierre de la lista, la lista no habrá terminado. Por ejemplo, podemos tener un código que se parezca a lo siguiente:

```
spam = ['apples',
    'oranges',
                    'bananas',
'cats']
print(spam)
```

Por supuesto, lo habitual es hacer que nuestras listas sean lo más legibles posible, como hicimos en el programa de la bola mágica.

También podemos dividir una instrucción en múltiples líneas usando el carácter \ de continuación de línea. Podemos pensar que el carácter \ quiere decir: "Esta instrucción continúa en la siguiente línea". La tabulación de la línea después del carácter \ no es significativa. Por ejemplo, el siguiente código Python es válido:

```
print('Four years and seven ' + \
      'months ago...')
```

Estos trucos son útiles cuando queremos redistribuir las líneas de código para hacer que un programa Python sea más legible.

TIPOS DE DATOS MUTABLES E INMUTABLES.

A pesar de sus similitudes, las listas y las cadenas son intrínsecamente distintas. Un valor tipo lista es un tipo de datos **mutable**, lo que significa que se le pueden añadir, quitar, o cambiar sus valores. Por el contrario, una cadena es **inmutable**, esto es, no puede cambiarse. Si intentamos cambiar un carácter individual dentro de una cadena, obtendremos un error `TypeError`:

```
>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment
```

La forma adecuada de "mutar" una cadena es usar el troceado y la concatenación para construir una *nueva* cadena copiando partes de la antigua. Escribe lo siguiente en el shell interactivo:

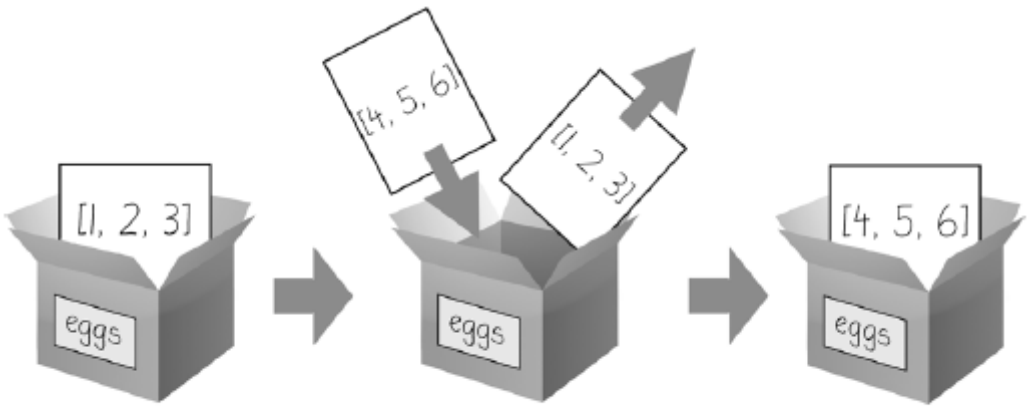
```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name
'Zophie a cat'
>>> newName
'Zophie the cat'
>>>
```

Aquí hemos usado `[0:7]` y `[8:12]` para referirnos a los caracteres que no queríamos reemplazar. Notar que la cadena `'Zophie a cat'` original no se ha visto modificada, porque las cadenas son inmutables.

Aunque un valor tipo lista es mutable, la segunda línea del siguiente código no modifica la lista `eggs`:

```
>>> eggs = [1, 2, 3]
>>> eggs = [4, 5, 6]
>>> eggs
[4, 5, 6]
```

Aquí no estamos cambiando el valor tipo lista almacenado en la variable `eggs`; en vez de eso, estamos sobreescribiendo el valor antiguo con un valor nuevo y completamente distinto. Este hecho se muestra en la figura.

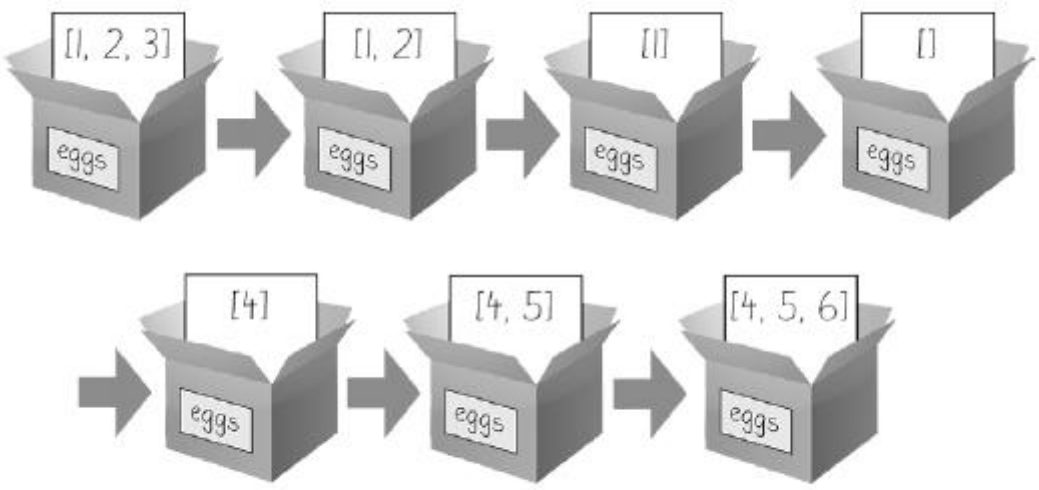


When eggs = [4, 5, 6] is executed, the contents of eggs are replaced with a new list value.

Si quisiéramos modificar la lista original almacenada en `eggs` para contener `[4, 5, 6]`, deberíamos hacer algo como esto:

```
>>> eggs = [1, 2, 3]
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append(4)
>>> eggs.append(5)
>>> eggs.append(6)
>>> eggs
[4, 5, 6]
```

En ambos ejemplos, la lista almacenada en la variable `eggs` termina con los mismos valores, a saber, `[4, 5, 6]`. La diferencia es que en este segundo ejemplo la lista ha cambiado, en lugar de ser sobreescrita. La figura muestra los pasos que hemos llevado a cabo en este segundo ejemplo:



The del statement and the append() method modify the same list value in place.

Cambiar el valor de un tipo de datos mutable, tal y como hemos hecho en el ejemplo previo usando las sentencias `del()` y `append()`, realmente ha cambiado el valor de lista, porque el valor de la variable no se ha visto reemplazado por un nuevo valor tipo lista.

Puede que hablar de tipos de datos mutables e inmutables nos parezca una distinción sin importancia, pero en la sección "Pasar referencias" más adelante veremos que llamar a funciones con argumentos mutables o inmutables puede producir comportamientos bien distintos. Pero antes de ello, vamos a presentar el tipo de datos tupla, que es la versión inmutable del tipo de datos lista.

EL TIPO DE DATOS TUPLA.

El tipo de datos **tupla** es casi idéntico al tipo de datos lista, excepto en dos aspectos. Primero, las tuplas se escriben con paréntesis, `(` y `)`, en lugar de hacerlo con corchetes. Por ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

Pero el aspecto fundamental en el que se diferencian las tuplas de las listas es que las tuplas, al igual que las cadenas, son inmutables. Los valores de las tuplas no se pueden modificar, anexar, ni quitar. Escribe lo siguiente en el shell interactivo, y observa cómo aparece el error `TypeError`:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

Si solo tenemos un valor en nuestra tupla, podemos indicar este hecho poniendo una coma después del valor dentro del paréntesis. En caso contrario, Python pensará que únicamente hemos tecleado un valor tipo cadena dentro de unos paréntesis normales. La coma es lo que le permite a Python saber que en realidad se trata de un valor de tipo tupla. (Al contrario que en otros lenguajes de programación, en Python es correcto escribir una última coma después del último elemento de una lista o tupla). Escribe las siguientes llamadas a la función `type()` en el shell interactivo para entender todo esto:

```
>>> type(('hello',))
<class 'tuple'>
>>> type(('hello'))
<class 'str'>
```

Podemos usar las tuplas para indicarle a alguien que esté leyendo nuestro código que no pretendemos cambiar una cierta secuencia de valores. Si queremos que una secuencia de valores no cambie nunca, debemos usar una tupla. Otra ventaja de usar tuplas en lugar de listas es que, como las tuplas son inmutables y su contenido no cambia, Python puede implementar ciertas optimizaciones que permiten que el código que usa tuplas sea algo más rápido que el código que usa listas.

CONVERTIR TIPOS CON LAS FUNCIONES LIST() Y TUPLE().

De la misma forma que `str(42)` devuelve `'42'`, esto es, la forma cadena del entero `42`, las funciones `list()` y `tuple()` devuelven las versiones tipo lista y tipo tupla de los valores que les pasemos. Escribe lo siguiente en el shell interactivo, y observa que el valor de retorno es de un tipo de datos distinto al valor pasado:

```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

Convertir una tupla en una lista puede ser útil cuando necesitemos una versión mutable de ese valor tipo tupla.

5.7. OTRAS FUNCIONALIDADES DE LAS LISTAS.

Los valores que forman los elementos de una lista también pueden especificarse usando otras variables, como por ejemplo:

```
>>> x = 1.0
>>> y = 1.5
>>> z = -2.2
>>> r = [x, y, z]
```

Como podemos comprobar, esto creará una lista de tres elementos con los valores `[1.0, 1.5, -2.2]`. En estos casos, es importante tener en cuenta que Python primero evaluará la expresión al lado derecho de la instrucción `r = [x, y, z]`, y a continuación, asignará ese valor a la variable de la izquierda. Es un error muy común pensar que como `r` es igual a `[x, y, z]`, si un programa cambia posteriormente el valor de `x`, el valor de `r` cambiará también. Esto es incorrecto. El valor de `r` seguirá siendo `[1.0, 1.5, -2.2]` y no cambiará aunque más tarde cambiemos el valor de `x`. A modo de ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> x = 1.0
>>> y = 1.5
>>> z = -2.2
>>> r = [x, y, z]
>>> r
[1.0, 1.5, -2.2]
>>> x = -3.6
>>> r
[1.0, 1.5, -2.2]
```

Los elementos de una lista también pueden calcularse mediante expresiones matemáticas, como por ejemplo:

```
>>> import math
>>> x = 1.0
>>> y = 1.5
>>> z = -2.2
>>> r = [2*x, x+y, z/math.sqrt(x**2+y**2)]
```

De nuevo, Python evaluará en primer lugar todas las expresiones a la derecha de la sentencia de asignación, y a continuación creará una lista a partir de los valores calculados.

Una vez creada una lista, probablemente querremos hacer cálculos con los elementos que contiene. Como ya deberíamos saber, podemos acceder a los elementos individuales de una lista a través de su índice, mediante `r[0]`, `r[1]`, `r[2]`, etc. Por ejemplo, si la lista `r = [2.0, -3.0, 1.5]` representa las tres componentes del vector tridimensional $\vec{r} = (x, y, z)$, podemos hallar su longitud $|\vec{r}| = \sqrt{x^2 + y^2 + z^2}$ como:

```
import math
x = 2.0
y = -3.0
z = -1.5
r = [x, y, z]
length = math.sqrt(r[0]**2 + r[1]**2 + r[2]**2)
print(length)
```

Sin embargo, una característica muy útil en Python es su capacidad para realizar operaciones sobre listas enteras de una sola vez. Por ejemplo, a menudo querremos saber la suma de los valores de una lista de números. Para ello, Python contiene una función integrada llamada `sum()` que nos permite calcular dichas sumas. Por ejemplo:

```
>>> r = [1.0, 1.5, -2.2]
>>> total = sum(r)
>>> total
0.2999999999999998
```

NOTA: Como ya habremos detectado, las operaciones con datos de tipo punto flotante no son infinitamente precisas. Al igual que en nuestras calculadoras de bolsillo, las operaciones en los ordenadores solo son precisas hasta un cierto número de cifras significativas (típicamente 16 cifras en los ordenadores modernos).

Esto significa que si asignamos el valor 1,0 a una variable de tipo punto flotante, puede que el ordenador guarde este valor como 0,9999999999999999. En muchos casos la diferencia no será muy importante, pero en otras situaciones esta diferencia puede ser crítica. Por ejemplo, imaginemos que nuestro programa debe hacer algo especial si un número es estrictamente menor que 1. En tal caso, la diferencia entre 1 y 0,9999999999999999 puede ser crucial. En los programas suelen aparecer errores y malfuncionamientos debido a este tipo de situaciones. Afortunadamente, hay una forma sencilla de evitarlos. Si la cantidad con la que estamos trabajando es genuinamente entera, debemos almacenarla en una variable de tipo entero. De esta forma estaremos seguros de que 1 significa 1. Los enteros no solo son precisos hasta 16 cifras significativas; son perfectamente precisos.

Otras funciones integradas que podemos manejar al trabajar con listas son las funciones `min()` y `max()`, que nos dan los valores máximo y mínimo de una lista numérica, respectivamente. También conocemos la función `len()`, que proporciona el número de elementos de la lista. A modo de ejemplo, vamos a obtener el valor máximo, el valor mínimo, y el valor promedio de los números almacenados en una lista:

```
>>> r = [5, 7, 2, 6]
>>> maxValue = max(r)
>>> minValue = min(r)
>>> average = sum(r)/len(r)
>>> maxValue
7
>>> minValue
2
>>> average
5.0
```

Otra función especial y ciertamente útil para las listas es la función `map()`, que nos permite aplicar funciones ordinarias, como `math.log` o `math.sqrt`, a todos los elementos de la lista. Por ejemplo, `map(math.log, r)` toma el logaritmo natural de cada uno de los elementos de una lista numérica, uno a uno. Técnicamente, la función `map` crea un objeto especializado en la memoria del ordenador, llamado **iterador**, que contiene todos los logaritmos. Normalmente querremos convertir este iterador en una nueva lista, para lo cual usaremos la función `list()`. A modo de ejemplo, consideremos el siguiente código:

```
>>> import math
>>> r = [1.0, 1.5, 2.2]
>>> logr = list(map(math.log, r))
>>> logr
[0.0, 0.4054651081081644, 0.7884573603642703]
```

5.8. REFERENCIAS.

Como ya sabemos de capítulos previos, las variables pueden almacenar valores de tipo entero y de tipo cadena. Escribe lo siguiente en el shell interactivo:

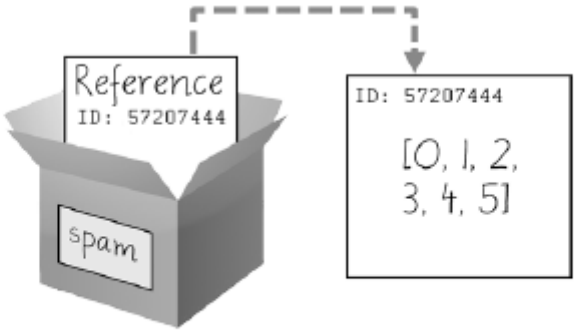
```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

Aquí asignamos el valor `42` a la variable `spam`, y a continuación copiamos el valor de `spam` y se lo asignamos a la variable `cheese`. Cuando posteriormente cambiamos el valor de `spam` a 100, esto no afecta al valor en la variable `cheese`. Esto es así porque `spam` y `cheese` son variables diferentes que almacenan valores distintos.

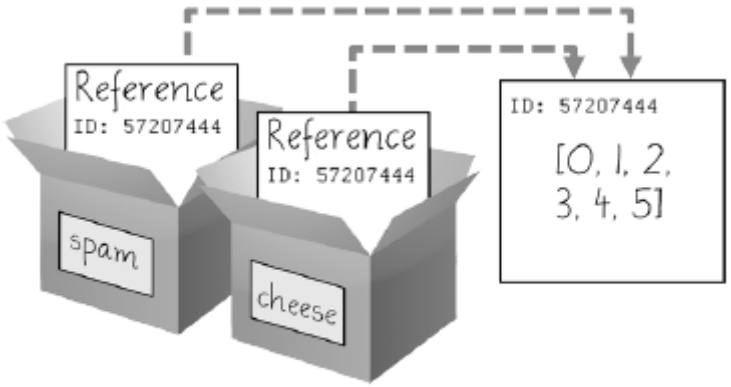
Sin embargo, las listas no funcionan de esta forma. Cuando a una variable le asignamos una lista, lo que realmente estamos haciendo es asignar a la variable una **referencia** a esa lista. Una referencia es un valor que apunta a la posición donde se encuentran unos ciertos bits de datos en la memoria de un ordenador, y una referencia a una lista es un valor que apunta a la posición de memoria donde se almacena esa lista. El siguiente código nos ayudará a entender esta distinción:

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
  >>> spam
  [0, 'Hello!', 2, 3, 4, 5]
  >>> cheese
  [0, 'Hello!', 2, 3, 4, 5]
```

Al principio, esto puede parecernos extraño: El código solo ha cambiado la lista `cheese`, pero este cambio ha afectado tanto a la lista `cheese` como a la lista `spam`. Vamos a explicar este comportamiento: Cuando en (1) creamos la lista, le asignamos una referencia que almacenamos en la variable `spam`. Pero la siguiente línea (2) copia únicamente la referencia a esa lista de la variable `spam` a la variable `cheese`, y no la lista en sí misma. Esto significa que los valores almacenados en las variables `spam` y `cheese` ahora se refieren a la misma lista. Solo hay una única lista subyacente, porque la lista en sí misma realmente nunca se copió. Por lo tanto, cuando en (3) modificamos el elemento en el índice 1 de la lista a la que se refiere la variable `cheese`, estamos modificando la misma lista a la que `spam` también se está refiriendo.



spam = [0, 1, 2, 3, 4, 5] stores a reference to a list, not the actual list.

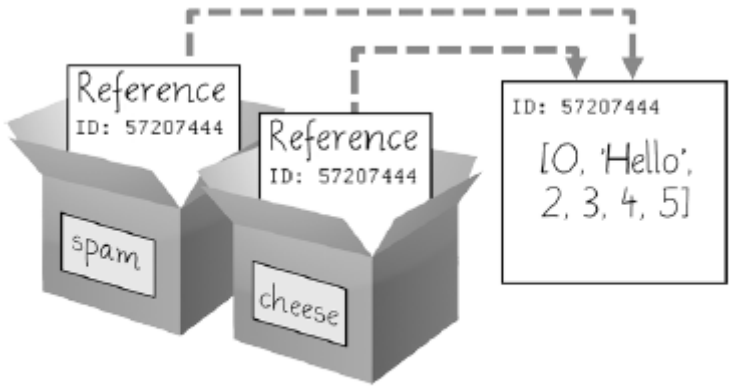


spam = cheese copies the reference, not the list.

Hasta ahora hemos dicho que podemos imaginar que las variables son como cajas que contienen valores. Las figuras que vemos arriba muestran que esta forma de entender las variables que contienen listas no es demasiado precisa, porque estas variables realmente no contienen listas, contienen las *referencias* a esas listas. (Estas referencias tendrán unos números de identificación que Python usa internamente para localizar en memoria las listas a las que apuntan, pero afortunadamente nosotros podemos ignorar todos estos detalles). La primera figura muestra qué ocurre cuando se asigna una lista a la variable `spam`, usando cajas como metáforas para las variables.

La segunda figura muestra lo que ocurre cuando la referencia almacenada en `spam` se copia a la variable `cheese`. En `cheese` solo se crea y se almacena una nueva referencia a la lista, no una lista nueva. Notar que ambas referencias apuntan a la misma lista.

Cuando alteramos la lista a la que se refiere `cheese`, la lista a la que se refiere spam también cambia, porque tanto `cheese` como `spam` apuntan, de hecho, a la misma lista. Esto se muestra en la tercera figura:



cheese[1] = 'Hello!' modifies the list that both variables refer to.

Por lo tanto, las variables contendrán referencias a valores tipo lista, y no los propios valores tipo lista. Por el contrario, y para el caso de los enteros y las cadenas, las variables simplemente contendrán el valor de tipo entero o de tipo cadena. Python siempre usará referencias cuando las variables deban almacenar valores de tipos de datos mutables, como las listas o los diccionarios. Para valores de tipo inmutable, como las cadenas, los enteros, o las tuplas, las variables de Python almacenarán el valor en sí mismo.

A pesar de que las variables de Python técnicamente contienen referencias a las listas y a los diccionarios, es habitual oír a la gente decir que la variable contiene una lista o un diccionario.

PASAR REFERENCIAS.

Las referencias son particularmente importantes para entender la forma en la que se le pasan argumentos a las funciones. Cuando se llama a una función, los valores de los argumentos se copian en las variables que actúan como parámetros de la función. En el caso de las listas (y diccionarios, aunque estos últimos los estudiaremos en el próximo capítulo), esto significa que se usa una copia de la referencia como valor para el parámetro. Para ver las consecuencias de todo esto, abre un nuevo archivo en el editor de archivos, escribe el siguiente código, y guárdalo como `passingReference.py`:

```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Notar que cuando se llama a la función `eggs()` no se usa un valor de retorno para asignar un nuevo valor a `spam`. En vez de eso, la función directamente modifica la propia lista in situ. Al ejecutarlo, este programa produce la siguiente salida:

```
[1, 2, 3, 'Hello']
```

Incuso aunque `spam` y `someParameter` contienen referencias distintas, ambas apuntan a la misma lista. Esta es la razón por la que la llamada al método `append('Hello')` dentro de la función afecta a la lista incluso después de que la llamada a la función haya retornado.

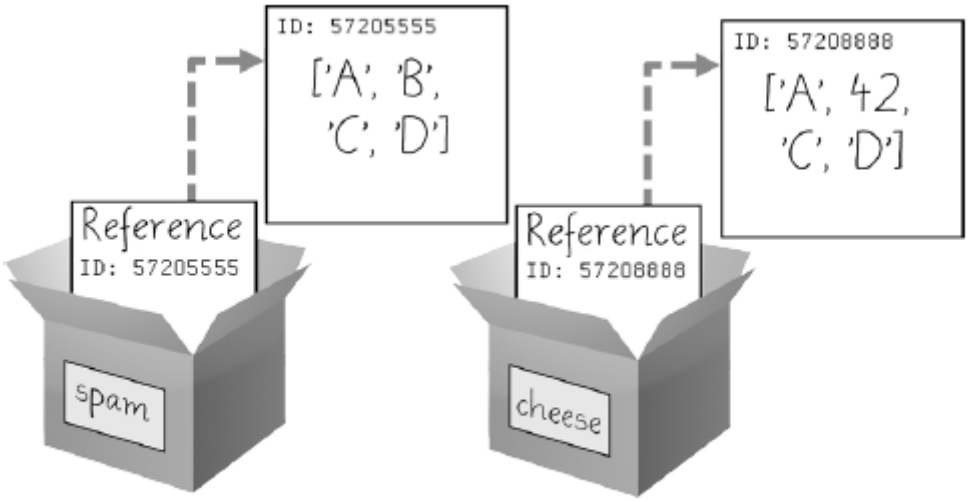
Es importante tener este comportamiento muy en cuenta. Si olvidamos que Python gestiona las variables de listas y diccionarios de esta forma, probablemente provoquemos errores y malfuncionamientos en nuestros programas.

LAS FUNCIONES COPY() Y DEEPCOPY() DEL MÓDULO COPY.

Aunque pasar referencias es habitualmente la forma más práctica de tratar con listas y diccionarios, si una función modifica la lista o el diccionario que se está pasando, puede que no queramos que tales cambios se reflejen en la lista o diccionario originales. Para ello, Python proporciona un módulo llamado `copy`, el cual incluye las funciones `copy()` y `deepcopy()`. La primera de ellas, `copy.copy()`, puede usarse para hacer un duplicado de un valor mutable como una lista o un diccionario, y no solo una copia de una referencia. Escribe lo siguiente en el shell interactivo:

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> cheese = copy.copy(spam)
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
['A', 42, 'C', 'D']
```

Ahora las variables `spam` y `cheese` se refieren a listas distintas, razón por la cual solo se ha visto modificada la lista en `cheese` cuando hemos asignado el entero `42` al índice `1`. Como podemos ver en la figura, los números identificadores de las referencias ya no son el mismo para las dos variables, porque las variables se refieren a dos listas independientes.



cheese = copy.copy(spam) creates a second list that can be modified independently of the first.

Si la lista que necesitamos copiar contiene otras listas dentro de ella, entonces debemos usar la función `copy.deepcopy()` en lugar de la función `copy.copy()`. La función `copy.deepcopy()` también copará esas listas internas.

5.9. EJERCICIOS DEL CAPÍTULO 5.

Ejercicio 5.1. Escribe una lista con los nombres de tus `4` o `5` mejores amigos. Escribe un programa que recorra la lista elemento tras elemento, y que imprima el mismo mensaje para cada uno de ellos. Por ejemplo:

```
Good morning Juan. Have a nice day.
```

```
Good morning Laura. Have a nice day.
Good morning Antonio. Have a nice day.
Good morning Elena. Have a nice day.
```

El programa debe funcionar igual de bien, independientemente del número de amigos en tu lista, y de si los nombres de la lista cambian. Guarda el programa como `Ejer5.1.py`.

Ejercicio 5.2. Digamos que queremos construir una lista de grados Celsius desde -50 a 200 en pasos de $2{,}5$ grados. Utiliza un bucle para añadiendo una elemento tras otro a una lista referenciada como `Celsius`. Guarda el programa como `Ejer5.2.py`.
NOTA: La función `range()` no admite argumentos flotantes. En este ejercicio, tal vez te convenga usar otro bucle en lugar de un bucle `for`.

Ejercicio 5.3. Listas numéricas.
a) Escribe una función que construya una lista con todos los números enteros desde uno hasta un millón.
b) En el mismo programa, escribe una función que recorra la lista e imprime por pantalla todos los números de la lista previa, un número por línea. (Si el programa tarda mucho en ejecutarse, presiona CTRL + C o cierra la ventana del shell).
c) Escribe una función que te permita sumar todos los elementos de la lista, e imprime el resultado por pantalla.
d) Escribe una función que te permita extraer de la lista los múltiplos de 13, y que los imprima por pantalla.
Guarda el programa que incluye todas estas funciones, y las llamadas necesarias para probarlas, como `Ejer5.3.py`.

Ejercicio 5.4. Suponer que comenzamos con la lista:

```
degrees = [0, 10, 20, 40, 100]
```

Escribe un programa que imprima en columna todos los elementos de la lista, y que al final muestre por pantalla cuántos elementos contiene esa lista. La salida debería ser similar a ésta:

```
list element (0) is: 0
list element (1) is: 10
list element (2) is: 20
list element (3) is: 40
list element (4) is: 100
The degrees list has 5 elements.
```

Cuidado: tu programa debe funcionar correctamente para una lista con un número de elementos cualquiera. Comprueba que funciona adecuadamente con varias listas distintas. Guarda el programa como `Ejer5.4.py`.

Ejercicio 5.5. El objetivo de este ejercicio es escribir un programa que imprima una tabla de valores para t e $y(t)$, donde:

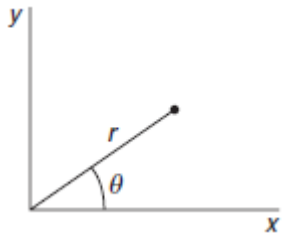
$$y(t) = v_o t - \frac{1}{2} g t^2$$

Usa $n+1$ valores de t igualmente espaciados entre los instantes de tiempo $t_0 = 0$ y $t_f = 2v_0/g$. n es el número de intervalos deseados entre esos instantes inicial y final (n es un parámetro que le debemos proporcionar al programa)[3], v_0 es la rapidez inicial (otro dato a proporcionar al programa), y $g = 9{,}81\ m/s^2$ es la aceleración de la gravedad en la Tierra.

[3] Notar que para definir n intervalos temporales, necesitamos $n+1$ instantes de tiempo que los delimiten. En efecto, para 2 intervalos, necesitamos 3 instantes de tiempo.

Para conseguirlo, haz lo siguiente: Primero, almacena los valores de t e y en dos listas, `t` e `y`. Después, obtén una tabla bien formateada recorriendo las dos listas con un bucle `for`. Guarda el programa como `Ejer5.5.py`.

Ejercicio 5.6.
(a) Escribe una función que reciba como argumento una lista con las coordenadas Cartesianas x e y de un punto sobre el plano. La función debe devolver una lista con las coordenadas polares r y θ de ese punto. De la figura, podemos ver que las fórmulas para pasar de coordenadas Cartesianas a polares son:



$$r = \sqrt{x^2 + y^2} \qquad \theta = \arctan\left(\frac{y}{x}\right)$$

(b) En el mismo archivo, escribe otra función que haga lo contrario, esto es, que reciba una lista con las coordenadas polares r y θ de un punto en el plano, y devuelva una lista con las coordenadas Cartesianas correspondientes. De la figura, es fácil ver que:

$$x = r\cos\theta \qquad y = r\sin\theta$$

Ejercicio 5.7. Código coma.
Imaginar que tenemos un valor tipo lista como éste:

```
spam = ['apples', 'bananas', 'tofu', 'cats']
```

Escribe una función que reciba este valor tipo lista como argumento y devuelva una cadena con todos los elementos separados por una coma y un espacio, con la palabra `and` insertada delante del último elemento. Por ejemplo, si pasamos la lista `spam` previa, la función debe retornar la cadena `'apples, bananas, tofu, and cats'`. Cuidado: tu función debe ser capaz de trabajar con cualquier valor tipo lista que se le pase. Guarda el programa como `Ejer5.7.py`.

Ejercicio 5.8. Imagina que tenemos una lista de temperaturas en grados Celsius de la forma:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
```

Escribe un programa que, para cada temperatura en grados Celsius, muestre la temperatura correspondiente en grados Fahrenheit, construyendo una tabla similar a:

```
 C     F
-20   -4.0
-15    5.0
-10   14.0
 -5   23.0
  0   32.0
  5   41.0
 10   50.0
 15   59.0
 20   68.0
 25   77.0
 30   86.0
 35   95.0
 40  104.0
```

Recuerda que la fórmula de conversión de Celsius a Fahrenheit ya la estudiamos en el capítulo 2. Guarda el programa como `Ejer5.8.py`.

Ejercicio 5.9. Imaginar que disponemos de una lista de listas (una **lista anidada**) donde cada valor en las listas internas es una cadena con un solo carácter, como por ejemplo:

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

(a) Extrae: 1) La letra `a`; 2) La lista `['d', 'e', 'f']`; 3) El último elemento `h`; 4) El elemento `d`. 5) ¿Qué elemento obtendrías escribiendo `q[-1][-2]`?
(b) Utiliza un bucle dentro de otro bucle para imprimir por pantalla de forma ordenada y uno tras otro todos los caracteres de la lista.

Ejercicio 5.10. El propósito de este ejercicio es escribir un programa que cree una lista de listas (una lista anidada) y la recorra elemento a elemento para imprimir por pantalla una tabla correctamente formateada. Para ello, computa dos listas, `t` e `y`, como explicamos en el ejercicio 5.5. A continuación, almacena ambas listas en una nueva lista *anidada* `ty1` de forma que `ty1[0]` se corresponda con la lista `t` y `ty1[1]` se corresponda con la lista `y`. Por último, consigue que el programa escriba una tabla con los valores t e y en dos columnas, iterando a lo largo de los datos en la lista `ty1`. Cada número debería escribirse redondeado a dos decimales; para ello usa la función integrada `round(floatNum, decimals)`, como en el siguiente ejemplo:

```
>>> import math
>>> round(math.pi, 4)
3.1416
```

Ejercicio 5.11. Escribe una función llamada `nested_sum` que reciba una lista de listas de enteros, y sume los elementos de todas las listas anidadas. Por ejemplo:

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

Ejercicio 5.12. Escribe una función llamada `middle` que reciba una lista y devuelva una nueva lista que contenga todos los elementos de la original, excepto en primero y el último. Por ejemplo:

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

Ejercicio 5.13. Escribe una función llamada `has_duplicates` que reciba una lista y devuelva `True` si hay algún elemento que aparezca más de una vez en la lista. La función no debe modificar la lista original.

Ejercicio 5.14. Dibujo de caracteres.
Digamos que disponemos de una lista de listas (lista anidada) donde cada valor en las listas internas es una cadena con un solo carácter, como por ejemplo:

```
grid = [['.', '.', '.', '.', '.', '.'],
        ['.', '0', '0', '.', '.', '.'],
        ['0', '0', '0', '0', '.', '.'],
        ['0', '0', '0', '0', '0', '.'],
        ['.', '0', '0', '0', '0', '0'],
        ['0', '0', '0', '0', '0', '.'],
        ['0', '0', '0', '0', '.', '.'],
        ['.', '0', '0', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.']]
```

Podemos considerar que `grid[x][y]` es el carácter en las coordenadas x e y del "dibujo" construido con caracteres de texto. El origen $(0, 0)$ está en la esquina superior izquierda, con la coordenada x aumentando

hacia la derecha, y la coordenada y aumentando hacia abajo. Copia la lista `grid` y escribe el código para mostrar por pantalla la imagen:

```
..00.00..
.0000000.
.0000000.
..00000..
...000...
....0....
```

(Pista: Necesitarás un bucle dentro de un bucle para imprimir `grid[0][0]`, luego `grid[1][0]`, luego `grid[2][0]`, y así sucesivamente, hasta `grid[8][0]`. Con esto habrás mostrado la primera línea, por lo que a continuación debemos empezar con una línea nueva. Por consiguiente, tu programa debería imprimir ahora `grid[0][1]`, luego `grid[1][1]`, después `grid[2][1]`, etc. Lo último que deberá imprimir tu programa es `grid[8][5]`. Recuerda también pasar como argumento a la función `print()` la palabra clave `end` si no quieres que se imprima una nueva línea automáticamente después de cada llamada a la función `print()`). Guarda el programa como `Ejer5.14.py`.

Ejercicio 5.15. Álgebra de vectores.
Suponer que tenemos dos vectores tridimensionales $\vec{a} = (a_1, a_2, a_3)$ y $\vec{b} = (b_1, b_2, b_3)$ y un número ordinario c (un escalar). El álgebra de vectores define 5 operaciones básicas con vectores:

1) Suma de vectores:

$$\vec{a} + \vec{b} = (a_1 + b_1, a_2 + b_2, a_3 + b_3)$$

2) Resta de vectores:

$$\vec{a} - \vec{b} = (a_1 - b_1, a_2 - b_2, a_3 - b_3)$$

3) Multiplicación de un vector por un escalar:

$$c\vec{a} = (ca_1, ca_2, ca_3)$$

4) Producto escalar de dos vectores:

$$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

5) Producto vectorial de dos vectores:

$$\vec{a} \times \vec{b} = (a_2 b_3 - a_3 b_2, \quad a_3 b_1 - a_1 b_3, \quad a_1 b_2 - a_2 b_1)$$

Escribe programa que defina una función para cada una de las operaciones vectoriales. Los vectores $\vec{a}$ y $\vec{b}$ se le pasan a la función como argumentos en forma de listas con tres elementos (las tres componentes del vector), y el escalar c como un argumento de tipo entero. Dependiendo de la operación a efectuar, la función devolverá un vector en forma de lista con tres elementos, o un entero (en el caso del producto escalar). En el propio programa, efectúa varias llamadas a las funciones definidas e imprime el resultado, para comprobar que todo funciona correctamente.

Ejercicio 5.16. Vacas y toros.
Escribe un programa que juegue a "Cows and Bulls" con el usuario. El juego funciona así: Se genera un número aleatorio de 4 dígitos, y se le pide al usuario que adivine ese número. Por cada dígito que el usuario haya acertado *en la posición correcta*, se tiene una "vaca" (cow). Por cada dígito que el usuario haya acertado en una posición incorrecta, se tiene un "toro" (bull). Cada vez que el usuario proponga un número, el program le dice cuántas "vacas" y "toros" tiene su número. Una vez que el usuario haya adivinado el número correcto, el juego termina, y le dice al usuario cuántos intentos ha necesitado para acertarlo.

Digamos que el ordenador ha generado el número 1038. Un ejemplo de interacción con el usuario sería el siguiente:

```
Welcome to the Cows and Bulls Game!
Enter a number:
>>> 1234
2 cows, 0 bulls
>>> 1243
1 cow, 1 bull
...
```

Ejercicio 5.17. Bingo.
Escribe un programa que simule un juego de bingo para dos jugadores. Cada jugador rellena una lista con 5 números de su elección, entre el 1 y el 30. Con otra lista numérica de 30 elementos representaremos el bombo virtual, e incluirá todos los números del 1 al 30. El programa debe de ir extrayendo aleatoriamente del bombo virtual los números que almacena, indicándoselos a los usuarios, y borrándolos de su lista (para que no vuelvan a salir). Cada vez que se saca un número del bombo, si ese número está en la lista de uno de los jugadores, debemos borrarlo de esa lista. Gana la partida el jugador que acabe antes con su lista de números vacía. Guarda el programa como `Ejer5.17.py`.

Ejercicio 5.18. Factores primos.
Supón que tenemos un número entero n y queremos hallar sus factores primos. Escribe una función que reciba como argumento el entero n a factorizar, y que devuelva una lista con los factores primos de ese entero. Guarda el programa como `Ejer5.18.py`.

Pista: Para hacer esto de forma relativamente fácil podemos dividir repetidamente n por todos los enteros k desde 2 hasta n, y comprobar si el resto es cero. Al encontrar un entero k que sea factor de n, debemos añadir ese entero a la lista de resultados, y hacer la división entera de n entre ese entero k (esto es, `n = n//k`), para extraer ese factor del número n y continuar comprobando posibles factores.

Ejercicio 5.19. Números primos.
El programa del ejemplo previo no es una forma muy eficiente de calcular números primos, porque comprueba cada número para ver si es divisible por cualquier número que sea menor que él. Podemos desarrollar un programa mucho más rápido para buscar números primos haciendo uso de las siguientes observaciones:

a) Por definición, un número n es primo si no tiene factores primos menores que n. Por lo tanto, solo necesitamos comprobar si es divisible por otros primos.
b) Si un número n no es primo y tiene un factor r, entonces $n = rs$, donde s también es un factor. Si $r \geq \sqrt{n}$ entonces $n = rs \geq \sqrt{n}s$, lo que implica que $s \leq \sqrt{n}$. En otras palabras, cualquier número no primo debe tener factores, y por consiguiente, también factores primos menores o iguales que $\sqrt{n}$. Por lo tanto, para determinar si un número es primo debemos comprobar sus factores primos solo hasta $\sqrt{n}$ ($\sqrt{n}$ incluido). Si no hay factores, entonces el número es primo.
c) Si encontramos un solo factor primo menor que $\sqrt{n}$ sabemos que el número no es primo, y no hay necesidad de comprobar nada más; podemos descartar ese número y comprobar otro.

Escribe un programa en Python que halle todos los números primos hasta 10000. Crea una lista para almacenar los números primos, la cual debe comenzar sólo con el número primo 2 dentro de ella. Entonces, para cada número n desde 3 hasta 10000, comprueba si el número es divisible por cualquiera de los primos que hay en la lista, hasta llegar a $\sqrt{n}$ (incluido). En cuanto encuentres un solo factor primo, puedes dejar de comprobar el resto de ellos, sabemos que n no es primo. Si no encontramos factores primos menores o iguales que $\sqrt{n}$, entonces n es primo y debemos añadirlo a la lista. Puedes imprimir la lista toda de una vez al final del programa, o puedes ir imprimiendo por pantalla los primos individuales conforme los vayas encontrando. Guarda el programa como `Ejer5.19.py`.

6. DICCIONARIOS Y ESTRUCTURAS DE DATOS.

En este capítulo presentamos el tipo de datos **diccionario**, el cual proporciona un mecanismo flexible de organizar y acceder a los datos. Combinando los diccionarios con nuestros conocimientos sobre listas, aprenderemos a crear **estructuras de datos** que permitan modelar objetos del mundo real.

6.1. EL TIPO DE DATOS DICCIONARIO.

Al igual que una lista, un **diccionario** es una colección de muchos valores. Pero al contrario que los índices de las listas, que obligatoriamente son enteros, los índices de los diccionarios pueden usar otros tipos de datos. A los índices de los diccionarios se les denomina **claves** (keys). Al conjunto formado por una clave y su correspondiente **valor** asociado (value) se le llama **par clave – valor**.

En Python, los diccionarios se escriben entre llaves, `{}`. Escribe lo siguiente en el shell interactivo:

```
myCat = {'size': 'fat', 'color': 'grey', 'age': 7}
```

Esta instrucción asigna un diccionario a la variable `myCat`. Las claves del diccionario son `'size'`, `'color'`, y `'age'`. Los valores para estas claves son `'fat'`, `'grey'`, y `7`, respectivamente. Podemos acceder a estos valores a través de sus claves:

```
>>> myCat['size']
'fat'
>>> 'My cat has ' + myCat['color'] + ' fur.'
'My cat has grey fur.'
```

Los diccionarios también pueden usar valores enteros como claves, de la misma forma que la listas usan enteros para sus índices. Sin embargo, estas claves no tienen por qué empezar por `0` ni ser consecutivas. De hecho, pueden ser un número cualquiera:

```
spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

DICCIONARIOS VS. LISTAS.

Al contrario que las listas, los **objetos** (items) que contienen los diccionarios (esto es, los pares clave – valor de los diccionarios) *no* están ordenados. El primer objeto de una lista llamada `spam` sería `spam[0]`. En cambio, no hay un "primer objeto" en un diccionario. Por lo tanto, mientras que el orden de los objetos es relevante para determinar si dos listas son la misma, el orden en el que se escriben los pares clave - valor de un diccionario no importa en absoluto. Escribe lo siguiente en el shell interactivo:

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

Como los diccionarios no están ordenados, no pueden cortarse como las listas.

Si intentamos acceder a una clave que no existe en un diccionario, obtendremos un mensaje de error de tipo `KeyError`, análogo al mensaje `IndexError` propio de las listas. Escribe lo siguiente en el shell interactivo, y observa el mensaje de error que aparece debido a que no hay una clave `'color'`:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

Aunque los diccionarios no están ordenados, el hecho de poder tener valores arbitrarios para las claves nos permite organizar nuestros datos de formas muy útiles. Por ejemplo, imaginar que queremos que nuestro programa guarde los datos sobre los cumpleaños de nuestros amigos. Podemos usar un diccionario con los nombres como claves y las fechas de los cumpleaños como valores. Abre una nueva ventana del editor de archivos y escribe el siguiente código. Guárdalo como `birthdays.py`:

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

  while True:
      print('Enter a name: (blank to quit)')
      name = input()
      if name == '':
          break

❷     if name in birthdays:
❸         print(birthdays[name] + ' is the birthday of ' + name)
      else:
          print('I do not have birthday information for ' + name)
          print('What is their birthday?')
          bday = input()
❹         birthdays[name] = bday
          print('Birthday database updated.')
```

En (1), creamos un diccionario inicial y lo guardamos en `birthdays`. Igual que en las listas, en (2) miramos si el nombre introducido existe como clave en nuestro diccionario, usando la palabra reservada `in`. Si ese nombre está en el diccionario, accedemos a su valor asociado usando corchetes (3). Si no está, podemos añadirlo usando la misma sintaxis basada en corchetes combinada con el operador de asignación (4).

Al ejecutar este programa, deberíamos obtener algo similar a esto:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Alex
I do not have birthday information for Alex
What is their birthday?
Dec 26
Birthday database updated.
Enter a name: (blank to quit)
Alex
Dec 26 is the birthday of Alex
Enter a name: (blank to quit)

>>>
```

Por supuesto, todos los datos que introduzcamos en el diccionario durante la ejecución del programa se borran cuando el programa termina. Aprenderemos a guardar datos de forma permanente (en archivos del disco duro) en capítulos posteriores.

LOS MÉTODOS KEYS(), VALUES(), Y ITEMS().

Hay tres *métodos* de los diccionarios que devuelven valores tipo lista de las claves del diccionario, de los valores del diccionario, o de ambos. Se trata de los métodos `keys()`, `values()`, y `items()`. Los valores que devuelven estos métodos no son listas auténticas: No se pueden modificar y no admiten el método `append()`. Sin embargo, estos tipos de datos (a saber, `dict_keys`, `dict_values`, y `dict_items`, respectivamente) pueden usarse en bucles `for` como otras listas cualesquiera. Para ver cómo funcionan estos métodos, escribe lo siguiente en el shell interactivo:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)


red
42
```

En este código, el bucle `for` itera (a través de la variable `v`, de "values") una vez sobre cada uno de los valores del diccionario `spam` (almacenados en la "lista" `spam.values()`). Un bucle `for` también puede iterar sobre las claves, o sobre los pares clave – valor ("items"):

```
>>> for k in spam.keys():
        print(k)


color
age
>>> for i in spam.items():
        print(i)


('color', 'red')
('age', 42)
```

Usando los métodos `keys()`, `values()`, y `items()`, un bucle `for` puede iterar sobre las claves, los valores, o los pares clave – valor de un diccionario, respectivamente. Notar que los valores en el valor `dict_items` devuelto por el método `items()` son tuplas formadas por la clave y el valor correspondiente.

Si queremos obtener una lista auténtica de alguno de estos métodos, le debemos pasar su valor de retorno a la función `list()`. Escribe lo siguiente en el shell interactivo:

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

La línea `list(spam.keys())` toma el valor `dict_keys` retornado por la función `keys()` y se lo pasa a la función `list()`, la cual devuelve el valor tipo lista `['color', 'age']`.

En un bucle `for` también podemos usar el truco de las asignaciones múltiples para asignar la clave y el valor a dos variables independientes. Escribe lo siguiente en el shell interactivo:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
        print('Key: ' + k + ', Value: ' + str(v))


Key: color, Value: red
Key: age, Value: 42
```

COMPROBAR SI UNA CLAVE O UN VALOR EXISTEN EN UN DICCIONARIO.

Recordemos del capítulo previo que los operadores `in` y `not in` nos permiten comprobar si un cierto valor existe en una lista. También podemos usar estos operadores para ver si una cierta clave o un cierto valor existen en un diccionario. Escribe lo siguiente en el shell interactivo:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

En el ejemplo previo, notar que `'color' in spam` es esencialmente una forma abreviada de escribir `'color' in spam.keys()`. De hecho, esto siempre podemos hacerlo *con las claves*: Si queremos comprobar si un valor es (o no es) una clave de un diccionario, podemos usar la palabra reservada `in` (o `not in`) con el propio valor tipo diccionario. (Ojo, esto no podemos hacerlo con los valores del diccionario).

EL MÉTODO GET().

Resulta un poco tedioso tener que comprobar si una cierta clave existe en un diccionario antes de acceder a esa clave. Afortunadamente, los diccionarios disponen del método `get()`, que recibe dos argumentos: la clave del valor a recuperar, y un valor alternativo a devolver si esa clave no existe en el diccionario. Escribe lo siguiente en el shell interactivo:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

Como no hay una clave `'eggs'` en el diccionario `picnicItems`, el método `get()` devuelve un valor por defecto de `0`. Si no hubiésemos usamos un `get()`, el código habría causado un mensaje de error como el del siguiente ejemplo:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
KeyError: 'eggs'
```

EL MÉTODO SETDEFAULT().

A menudo tendremos que darle un valor a una cierta clave de un diccionario solo si esa clave no existía previamente. El código se parecería a lo siguiente:

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

El método `setdefault()` ofrece una forma alternativa de hacer esto en una sola línea de código. El primer argumento que se le pasa al método es la clave a comprobar, y el segundo argumento en el valor que queremos asociarle a esa clave si esa clave no existe. Si la clave ya existe, el método `setdefault()` retorna el valor que ya tenía la clave, y su valor no cambia. Escribe lo siguiente en el shell interactivo:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'name': 'Pooka', 'age': 5, 'color': 'black'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'name': 'Pooka', 'age': 5, 'color': 'black'}
```

La primera vez que llamamos al método `setdefault()`, el diccionario en `spam` cambia de `{'name': 'Pooka', 'age': 5}` a `{'name': 'Pooka', 'age': 5, 'color': 'black'}`. Además, el método devuelve el valor `'black'`, porque éste es ahora el valor asociado a la clave `'color'`. A continuación, al llamar a `spam.setdefault('color', 'white')`, el valor de esa clave no se cambia a `'white'`, porque en `spam` ya había una clave llamada `'color'`.

El método `setdefault()` es un buen atajo para asegurarnos de que una clave existe. He aquí un pequeño programa que cuenta el número de apariciones de cada letra en una cadena dada. Abre el editor de archivos y escribe el siguiente código, guardándolo como `characterCount.py`:

```
message = 'It was a bright cold day in April, and
          the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

El programa itera sobre cada carácter dentro de la cadena almacenada en la variable `message`, y cuenta cuántas veces aparece cada carácter en la cadena. La llamada al método `setdefault()` asegura que la clave `character` está en el diccionario `count` (con un valor por defecto de `0`). De esta forma, el programa no lanza un error `keyError` cuando ejecuta `count[character] = count[character] + 1`. Al ejecutar el programa, la salida debería ser:

```
{'I': 1, 't': 6, ' ': 13, 'w': 2, 'a': 4, 's': 3, 'b': 1, 'r': 5, 'i': 6
, 'g': 2, 'h': 3, 'c': 3, 'o': 2, 'l': 3, 'd': 3, 'y': 1, 'n': 4, 'A': 1
, 'p': 1, ',': 1, 'e': 5, 'k': 2, '.': 1}
```

En la salida del programa podemos ver que la letra `c` minúscula aparece `3` veces, el carácter de espaciado aparece `13` veces, y la letra `A` mayúscula aparece `1` vez. Este programa funciona independientemente de la

cadena que haya dentro de la variable `message`, incluso aunque tenga una longitud de un millón de caracteres.

6.2. IMPRESIÓN CON FORMATO.

Si importamos el módulo `pprint` (de "pretty printing"), tendremos acceso a las funciones `pprint()` y `pformat()`, que nos permitirán una "impresión bonita" de los valores de un diccionario. Esto es muy útil cuando queremos mostrar los objetos que contiene un diccionario de forma más limpia, en comparación a como lo hace la función `print()`. Modifica el programa `characterCount.py` previo y guárdalo como `prettyCharacterCount.py`.

```
import pprint
message = 'It was a bright cold day in April, and
          the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

Esta vez, al ejecutar el programa, la salida se muestra de forma mucho más limpia, con las distintas claves ordenadas:

```
{' ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
 'w': 2,
 'y': 1}
```

La función `pprint.pprint()` es especialmente útil cuando el propio diccionario contiene listas o diccionarios anidados.

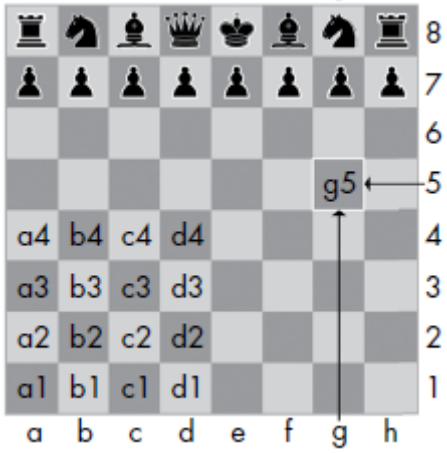
Si queremos guerdar el texto formateado en una cadena, en vez de mostrarlo por pantalla, debemos llamar a la función `pprint.pformat()`. Así, estas dos líneas de código son equivalentes:

```
pprint.pprint(someDictionaryValue)
print(pprint.pformat(someDictionaryValue))
```

6.3. USAR ESTRUCTURAS DE DATOS PARA MODELAR OBJETOS DEL MUNDO REAL.

Incluso antes de que existiera Internet, ya era posible jugar al ajedrez con alguien que estuviese en la otra punta del mundo. Cada jugador desplegaba un tablero de ajedrez en su casa, y ambos se comunicaban por vía postal cuáles eran los movimientos que realizaban. Para poder hacerlo, los jugadores necesitaban una forma de describir sin ambigüedad el estado del tablero y sus respectivos movimientos.

En la notación de ajedrez algebraica, las casillas del tablero se identifican mediante unas coordenadas basadas en letras y números, como muestra la figura:



Por su parte, las piezas se identifican mediante letras: K para el rey (king), Q para la reina (queen), R para la torre (rook), B para el alfil (bishop), N para el caballo (knight), y P para el peón (pawn). La descripción de un movimiento implica usar la letra de la pieza y las coordenadas de su destino. Dos movimientos como éstos describen lo que ocurre en un turno (con las piezas blancas moviendo primero). Por ejemplo, la notación $2.\ Nf3\ Bc6$ indica que, en el segundo turno de la partida, el jugador de las piezas blancas mueve el caballo a la casilla $f3$, y que el jugador de las piezas negras mueve el alfil a la casilla $c6$.

Con esta notación, es posible describir sin ambigüedad un juego de ajedrez sin necesidad de que los jugadores estén cara a cara delante del mismo tablero. De esta forma, se puede jugar una partida de ajedrez desde extremos opuestos del mundo. De hecho, ni siquiera es necesario disponer físicamente de un tablero si tenemos una buena memoria: Basta con leer los movimientos que nuestro oponente nos envía por correo, y actualizar la configuración del tablero en nuestra cabeza.

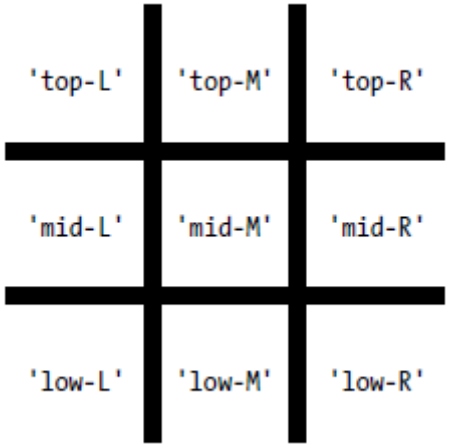
Los ordenadores sí que tienen buenas memorias. Un programa de un ordenador moderno puede contener miles de millones de cadenas como `'2. Nf3 Bc6'`. Así es como los ordenadores pueden jugar al ajedrez incluso sin disponer de un tablero físico. Los ordenadores modelan datos para representar un tablero de ajedrez, y nosotros podemos escribir programas para trabajar con este modelo.

Aquí es donde las listas y los diccionarios entran en juego. Podemos usarlos para modelar cosas del mundo real, como por ejemplo, tableros de ajedrez. A modo de ejemplo, vamos a hacer un juego que es algo más simple que un ajedrez: un juego de las tres en raya (tic-tac-toe).

UN TABLERO PARA LAS TRES EN RAYA.

El tablero de las tres en raya se parece a un gran símbolo de almohadilla (#) con nueve casillas, cada una de las cuales puede contener una X, una O, o estar en blanco. Para representar este tablero con un diccionario, podemos asignar a cada casilla una clave de tipo cadena, como muestra la figura.

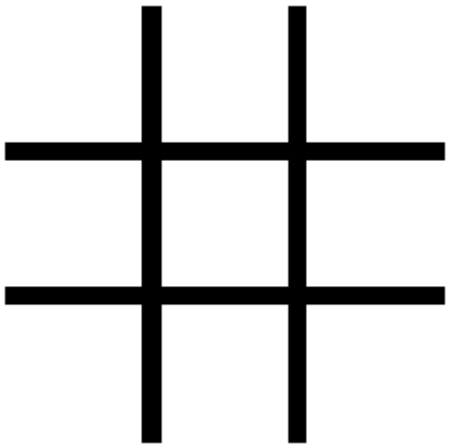
Además, podemos usar valores de tipo cadena para representar qué hay en cada una de las casillas del tablero: `'X'`, `'O'`, ó `' '` (un carácter en blanco). Por consiguiente, necesitaremos almacenar nueve cadenas. Para ello, podemos usar un diccionario: El valor cadena con la clave `'top-L'` (top - Left) representará el estado de la casilla en la fila superior y a la izquierda, el valor cadena con la clave `'low-R'` (low - Right) representará el estado de la casilla en la fila inferior y a la derecha, el valor cadena con la clave `'mid-M'` (middle - Middle) representará el estado de la casilla en la fila central y en el centro, etc.



Este diccionario es una estructura de datos que representa un tablero para las tres en raya. Vamos a almacenar este tablero (board) en la forma de un diccionario dentro de una variable llamada `theBoard`. Abre una nueva ventana del editor de archivos, escribe el siguiente código fuente, y guárdalo como `ticTacToe.py`:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

La estructura de datos almacenada en la variable `theBoard` representa el tablero mostrado en la figura:

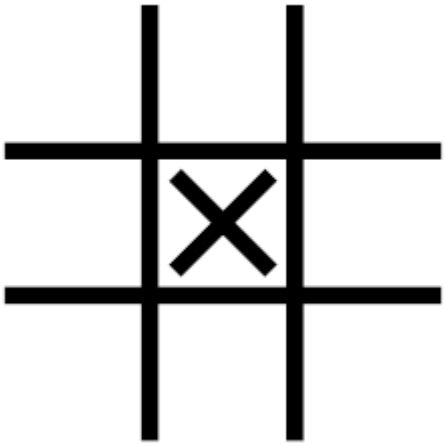


Como el valor para cada clave del diccionario `theBoard` es una cadena con un solo espacio en blanco, este diccionario representa un tablero completamente vacío.

Si el jugador *X* mueve en primer lugar y elige la casilla central, podríamos representar ese tablero con el siguiente diccionario:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

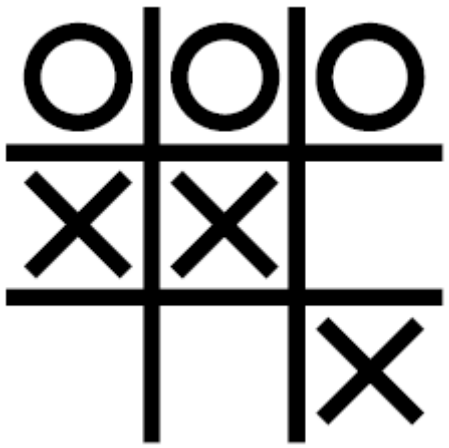
En este caso, la estructura de datos `theBoard` representaría el tablero mostrado en la figura:



Un ejemplo de tablero en el que el jugador *O* haya ganado ubicando tres *O*'s en la fila superior se parecería al siguiente:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

Esta estructura de datos representaría el tablero de la figura:



Por supuesto, el jugador solo ve qué es lo que se imprime por pantalla, y no los contenidos de las variables. Vamos a crear una función que imprima el diccionario del tablero en pantalla. Añade lo siguiente al programa `ticTacToe.py`:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])

printBoard(theBoard)
```

Cuando ejecutemos este programa, la función `printBoard()` imprimará una tablero de las tres en raya vacío:

```
 | |
-+-+-
 | |
-+-+-
 | |
```

La función `printBoard()` puede manejar cualquier estructura de datos de las tres en raya que le pasemos como argumento. Prueba a cambiar el código de la siguiente forma:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])

printBoard(theBoard)
```

Si ahora ejecutamos este programa modificado, el tablero que imprimiremos por pantalla será:

```
O|O|O
-+-+-
X|X|
-+-+-
 | |X
```

Como hemos creado una estructura de datos para representar un tablero de las tres en raya y hemos escrito un código en la función `printBoard()` para interpretar esa estructura de datos, ahora tenemos un programa que "modela" un tablero de las tres en raya. Podríamos haber organizado nuestra estructura de datos de forma distinta (por ejemplo, podríamos haber usado claves como `'TOP-LEFT'` en lugar de `'top-L'`), pero mientras el código que escribamos trabaje con la estructura de datos por la que nos hayamos decantado, tendremos un programa que funciona correctamente.

Por ejemplo, la función `printBoard()` espera que la estructura de datos del tablero de las tres en raya sea un diccionario con claves para las nueve casillas del tablero. Si el diccionario que le pasamos carece, digamos, de la clave `'mid-L'`, el programa ya no funcionará correctamente.

```
O|O|O
-+-+-
Traceback (most recent call last):
  File "C:/Users/Usuario/Desktop/1.py", line 10, in <module>
    printBoard(theBoard)
  File "C:/Users/Usuario/Desktop/1.py", line 7, in printBoard
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
KeyError: 'mid-L'
```

Ahora vamos a añadir el código que permite a los jugadores introducir sus movimientos. Modifica el programa `ticTacToe.py` para añadir lo siguiente:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])

turn = 'X'
for i in range(9):
❶   printBoard(theBoard)
    print('Turn for ' + turn + '. Move on which space?')
❷   move = input()
❸   theBoard[move] = turn
❹   if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'

printBoard(theBoard)
```

El nuevo código imprime el tablero al comienzo de cada uno de los 9 turnos posibles (1), obtiene el movimiento del jugador al que le toca mover (2), actualiza el tablero de acuerdo con este movimiento (3), y cambia el turno al otro jugador (4) antes de empezar con el siguiente turno. Al ejecutar este programa, funcionará de forma similar a lo mostrado en la figura:

```
 | |
-+-+-
 | |
-+-+-
 | |
Turn for X. Move on which space?
mid-M
 | |
-+-+-
 |X|
-+-+-
 | |
Turn for O. Move on which space?
top-L
O| |
-+-+-
 |X|
-+-+-
 | |
Turn for X. Move on which space?
mid-L
O| |
-+-+-
X|X|
-+-+-
 | |
```

(continúa…)

```
Turn for O. Move on which space?
top-R
O|X|O
-+-+-
X|X|O
-+-+-
 | |
Turn for X. Move on which space?
low-M
O|X|O
-+-+-
X|X|O
-+-+-
 |X|
Turn for O. Move on which space?
low-R
O|X|O
-+-+-
X|X|O
-+-+-
 |X|O
Turn for X. Move on which space?
```

Como podemos comprobar, este programa no es un juego de las tres en raya acabado (por ejemplo, no comprueba si un jugador ha ganado la partida), pero es lo suficientemente completo como para mostrar cómo podemos usar estructuras de datos en programas para modelar objetos del mundo real.

DICCIONARIOS Y LISTAS ANIDADOS.

Modelar un tablero para el juego de las tres en raya ha sido bastante sencillo: el tablero solo necesitaba un único valor de tipo diccionario con nueve pares clave – valor. Conforme necesitemos modelar cosas más complicadas, probablemente tengamos que usar diccionarios y listas que contengan otros diccionarios y listas.

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):
    numBrought = 0
❶   for k, v in guests.items():
❷       numBrought = numBrought + v.get(item, 0)
    return numBrought

print ('Number of things being brought:')
print (' - Apples: ' + str(totalBrought(allGuests, 'apples')))
print (' - Cups: ' + str(totalBrought(allGuests, 'cups')))
print (' - Cakes: ' + str(totalBrought(allGuests, 'cakes')))
print (' - Ham sandwiches: ' + str(totalBrought(allGuests, 'ham sandwiches')))
print (' - Apple pies: ' + str(totalBrought(allGuests, 'apple pies')))
```

Las listas son útiles para contener una serie *ordenada* de valores, y los diccionarios son útiles para asociar claves y valores. Por ejemplo, aquí tenemos un programa que usa un diccionario que contienen otros diccionarios para ver quién trae qué a un picnic en el campo. La función `totalBrought()` puede leer esta estructura de datos y calcular el número total de unidades de un cierto artículo traídas por todos los asistentes al picnic.

Dentro de la función `totalBrought()`, el bucle `for` itera sobre los pares clave – valor en el diccionario `guests` (1). Dentro del bucle, la cadena del nombre del asistente se asigna a la variable `k`, y el diccionario de los artículos que trae cada asistente se asigna a la variable `v`. Si el artículo existe como clave en este diccionario, su valor (esto es, la cantidad) se suma a `numBrought` (2). Si ese artículo no existe como clave, el método `get()` devuelve un `0` que será sumado a `numBrought`.

La salida del programa se parecerá a lo siguiente:

```
Number of things being brought:
 - Apples: 7
 - Cups: 3
 - Cakes: 0
 - Ham sandwiches: 3
 - Apple pies: 1
```

Este programa puede parecernos un poco complicado para lo que realmente hace. Pero notar que la misma función `totalBrought()` podría manejar fácilmente un diccionario que contuviese, digamos, cinco mil invitados al picnic, cada uno de ellos llevando decenas de artículos. Por lo tanto, tener esta información en una estructura de datos de tipo diccionario anidado, junto con la función `totalBrought()`, nos permitiría contabilizar todos los artículos disponibles, ahorrándonos un montón de tiempo.

Disponemos de total libertad para modelar las cosas con las estructuras de datos que mejor nos parezca, siempre que el resto del código de nuestro programa trabaje de forma consistente con el modelo de datos por el que nos hayamos decantado. En nuestros primeros programas no debemos preocuparnos demasiado por identificar cuál es la forma más "correcta" de modelar ciertos datos. Conforme adquiramos más experiencia, se nos irán ocurriendo modelos más eficientes. Lo importante es que la estructura de datos en la que pensemos funcione correctamente, y que permita a nuestro programa hacer lo que se le pide.

6.4. EJERCICIOS DEL CAPÍTULO 6.

Ejercicio 6.1. Inventario para un juego de fantasía.
Imagina que estás creando un videojuego de fantasía. La estructura de datos para modelar el inventario de objetos del jugador será un diccionario donde las claves son valores de tipo cadena que describen el objeto en el inventario, y el valor es un entero que detalla cuántos de esos objetos posee el jugador. Por ejemplo, el diccionario `{'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}` indica que el jugador dispone de 1 cuerda (rope), 6 antorchas (torch), 42 monedas de oro (gold coin), 1 puñal (dagger), y 12 flechas (arrow).

Escribe una función llamada `displayInventory()` que reciba cualquier inventario posible y lo muestre de la siguiente manera:

```
Inventory:
1 rope
6 torch
42 gold coin
1 dagger
12 arrow
Total number of items: 62
```

(Pista: Utiliza un bucle para iterar a través de todas las claves del diccionario). Guarda el programa como `Ejer6.1.py`.

Ejercicio 6.2. Crea un diccionario que almacene las capitales de 10 países. A continuación, recorre el diccionario para imprimir una salida como la siguiente:

```
The capital city of France is Paris.
The capital city of Italy is Rome.
The capital city of Kenya is Nairobi.
The capital city of Vietman is Hanoi.
…
```

Ejercicio 6.3. Imagina que, en nuestro juego de fantasía del ejercicio 6.1, el botín que obtienes al saquear la guarida de un dragón al que has vencido viene representado por una *lista* de cadenas como la siguiente:

```
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
```

Escribe una función llamada `addToInventory(inventory, addedItems)` en la que el parámetro `inventory` sea un *diccionario* que represente el inventario de objetos del jugador (como en el ejercicio 6.1) y el parámetro `addedItems` sea una *lista* como `dragonLoot`.

La función `addToInventory()` debería devolver un *diccionario* que represente el inventario actualizado. Notar que la lista `addedItems` puede contener múltiples objetos iguales. Tu código debería parecerse a lo siguiente:

```
def addToInventory(inventory, addedItems):
    # your code goes here.

inv = {'gold coin': 42, 'rope': 1}
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
inv = addToInventory(inv, dragonLoot)
displayInventory(inv)
```

Este programa (con la función `displayInventory()` que escribiste en el ejercicio 6.1) debería mostrar la siguiente salida:

```
Inventory
45 gold coin
1 rope
1 ruby
1 dagger

Total number of items: 48
```

Guarda el programa como `Ejer6.3.py`.

Ejercicio 6.4. Nota media.
Crea un programa que permita almacenar en un diccionario las notas de un grupo de alumnos. Las claves del diccionario serán los nombres de los alumnos. El valor de cada clave será la nota de ese alumno. El programa deberá ir pidiendo nombres de alumnos y sus respectivas notas, hasta que el usuario ya no quiera introducir ningún alumno más. Una vez introducidos los datos, el programa deberá imprimir por pantalla el conjunto de los alumnos, junto con las notas asociadas. Además, el programa deberá obtener e imprimir la nota media conjunta de todos los alumnos. Guarda el programa como `Ejer6.4.py`.

Ejercicio 6.5. Polinomios como diccionarios.
Podemos usar diccionarios con enteros como claves para representar polinomios. Por ejemplo, el polinomio:

$$p(x) = -1 + x^2 + 3x^7$$

, podría representarse mediante el diccionario:

```
p = {0: -1, 2: 1, 7: 3}
```

, donde la clave representa la potencia de la variable x, y el valor el coeficiente de esa potencia.

También podríamos modelar polinomios con listas, pero sería menos práctico: tendríamos que rellenar todos los coeficientes que sean cero, porque el índice de la lista representaría la potencia:

```
p = [-1, 0, 1, 0, 0, 0, 0, 3]
```

Asumiendo que vamos a modelar un polinomio con un diccionario, escribe una función que reciba como argumentos un diccionario que represente un polinomio y un valor de x. La función debe devolver el valor al que se evalúa el polinomio para ese valor de x. Por ejemplo, para $x = 1$, el polinomio $p(x) = -1 + x^2 + 3x^7$ debería evaluarse a $p(x) = -1 + (1)^2 + 3(1)^7 = -1 + 1 + 3 = 3$. Guarda el programa como `Ejer6.5.py`.

Ejercicio 6.6. Escribe un programa que comience con una *lista de diccionarios*. Cada diccionario de la lista representará a una persona, y las claves serán su nombre, su edad (en años), su peso (en kilogramos), y su altura (en metros). La lista contendrá al menos tres personas (tres diccionarios). Recorre la lista de diccionarios para imprimir adecuadamente (correctamente formateados) todos los datos almacenados sobre cada persona. Guarda el programa como `Ejer6.6.py`.

Ejercicio 6.7. Calificación.
Crea tres diccionarios, uno por cada alumno (por ejemplo, Lloyd, Alice, y Tyler):

```
peter = {
  'name': 'Peter',
  'homework': [90.0,97.0,75.0,92.0],
  'quizzes': [88.0,40.0,94.0],
  'tests': [75.0,90.0]
}
alice = {
  'name': 'Alice',
  'homework': [100.0, 92.0, 98.0, 100.0],
  'quizzes': [82.0, 83.0, 91.0],
  'tests': [89.0, 97.0]
}
tyler = {
  'name': 'Tyler',
  'homework': [0.0, 87.0, 75.0, 22.0],
  'quizzes': [0.0, 75.0, 78.0],
  'tests': [100.0, 100.0]
}
```

Cada diccionario tiene 4 claves de tipo cadena, a saber, `'name'`, `'homework'`, `'quizzes'`, y `'tests'`. El valor de la primera clave (`'name'`) es la cadena con el nombre del alumno, y los valores de las siguientes claves son listas con las notas que ese alumno ha obtenido en sus deberes (`'homework'`), en sus cuestionarios (`'quizzes'`), y en los exámenes (`'tests'`). Haz lo siguiente:
(a) Escribe una función llamada `average` que reciba una lista de números y obtenga su media.
(b) Escribe una función `studentAverage` que reciba el nombre de un alumno, y obtenga la media de todas sus notas en los deberes, cuestionarios, y exámenes. (Los deberes y cuestionarios valen el 20% de la nota final cada uno, y los exámenes el 60% restante).
(c) Escribe una función llamada `getStudentGrade` que reciba como argumento una nota (del 0 al 100) y devuelva su calificación final. Si esa nota es mayor o igual que 90, la función devuelve A. Si es mayor o igual que 80 (pero menor que 90) la función devuelve B. Si es mayor o igual que 70 (pero menor que 80) la

función devuelve C. Si es mayor o igual que 60 (pero menor que 70) la función devuelve D. en otro caso, devuelve F.
(d) Por último, escribe una función llamada `classGrades` que devuelva un diccionario con la calificación (A, B, C, D, o F) de todos los alumnos de la clase. La clave será el nombre del alumno, y el valor la calificación.

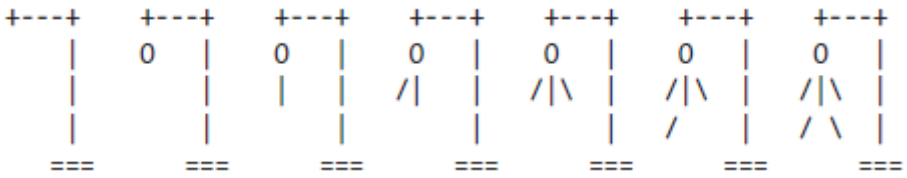
Ejercicio 6.8. Diccionario anidado.
Queremos un diccionario (llamémosle `fruits`) cuya clave sea el nombre de cada fruta en nuestra tienda (por ejemplo, `'bananas'`, `'apples'`, `'oranges'`, y `'tomatoes'`). El valor de cada clave será un diccionario donde las claves serán `'stock'` y `'unitPrice'`. Los valores de estas claves, para cada tipo de fruta, serán el número de frutas en stock de ese tipo, y el precio de cada unidad. Crea un programa que le permita al usuario introducir las frutas que hay en su tienda, su stock, y su precio unitario. Una vez creada la lista, el programa debe permitir calcular cuánto dinero conseguiría el usuario si consiguiese vender todos los artículos de su tienda. Guarda el programa como `Ejer6.8.py`.

Ejercicio 6.9. Escribe un programa que le pida al usuario un entero positivo n. El programa debe construir un diccionario donde las claves sean todos los enteros positivos desde 1 hasta n, y donde el valor sea el cuadrado de ese número. Por ejemplo, para $n = 4$, el programa debería devolver:

```
{1: 1, 2: 4, 3: 9, 4: 16}
```

Ejercicio 6.10. Juego del ahorcado.
Vamos a programas el juego del ahorcado. El ordenador escoge una palabra secreta al azar (de una lista de palabras preconstruida), y el jugador debe intentar acertarla, una letra tras otra. Cada vez que el jugador falle una letra, el ordenador muestra una nueva imagen de una figura en una horca. Sigue estos pasos para programarlo.

a) Crea los gráficos de las siete versiones de la horca. Estos gráficos los dibujaremos con caracteres del teclado impresos por pantalla, como muestra la figura. (A este tipo de gráficos se les denomina **ASCII art**).



Para almacenar los siete dibujos de la horca vamos a crear una tupla, donde cada elemento será una cadena que represente cada versión de la horca:

```
HANGMAN = (
"""
 +---+
     |
     |
     |
    ===
""",
"""
 +---+
 O   |
     |
     |
    ===
""",
.
.
.
```

```
"""
 +---+
 O   |
/|\  |
/ \  |
    ===
""")
```

Notar que cada cadena de la tupla la hemos escrito entre tres comillas dobles, en lugar de hacerlo entre comillas simples. La razón para ello quedará clara en el próximo capítulo.

b) Creamos las constantes:
- La primera, `MAX_WRONG`, almacenará el número total de fallos que puede tener el jugador (¿sabes cómo calcularla?).
- La segunda será una tupla, `WORDS`, con el conjunto de palabras posibles entre las que el programa elegirá la palabra secreta a adivinar. Elige un conjunto de 5 ó 6 palabras en MAYÚSCULAS para guardar en esta tupla.

c) Inicializamos las variables:
- En la variable `word` almacenamos una de las palabras de la tupla `WORDS` elegida al azar.
- En la variable `so_far` indicamos qué letras de la palabra secreta ha acertado el jugador. Esta variable comienza almacenando una cadena con tantos guiones como letras tiene la palabra secreta. Cuando el usuario acierte una letra, el guión en esa posición será sustituido por la letra acertada.
- La variable `wrong` lleva la cuenta del número de errores cometidos por el jugador. La inicializamos a cero.
- La lista `used` irá almacenando las letras que el jugador ya ha usado. La inicializamos vacía.

d) Creamos el bucle principal.
El bucle principal estará iterando mientras el jugador no haya acumulado el número máximo de fallos posibles y mientras aún no haya adivinado la palabra secreta. A cada iteración, el bucle imprime por pantalla la figura actual para la horca (que será `HANGMAN[wrong]`, ¿entiendes por qué?), imprime las letras que el jugador ya ha usado, y el estado actual de la palabra secreta (esto es, de la variable `so_far`).

e) Obtenemos la letra propuesta por el jugador.
Le pedimos al jugador la letra (que almacenamos en la variable `guess`), y la convertimos a mayúsculas (`guess = guess.upper()`) para asegurarnos de que la encontraremos en la palabra secreta. Antes de nada, comprobamos de forma continuada que el jugador no haya usado ya esa letra, buscándola en `used`. Si el jugador ya usó esa letra, se lo indicamos y le pedimos otra letra. Una vez el usuario ha proporcionado una letra no usada antes, la convertimos a mayúsculas y la añadimos a la lista de letras ya usadas.

f) Comprobamos la letra.
Ahora comprobamos si la letra proporcionada está en la palabra secreta.

De ser así, se lo hacemos saber al usuario, y a continuación, creamos una nueva versión de `so_far` para incluir esta letra en todos los lugares donde aparezca en la palabra sereta. Para ello, creamos una nueva cadena vacía, `new`. Iteramos con un bucle `for` a lo largo de la palabra secreta, comprobando carácter a carácter si la letra del jugador está presente. De ser así, concatenamos a new esa letra. De no ser así, concatenamos a `new` la posición actual de la antigua cadena `so_far`. Al salir del bucle, habremos recorrido la palabra secreta, y en new tenemos la nueva versión de `so_far`, por lo tanto, asignamos a `so_far` el valor de `new`.

Pero si la letra del usuario no está en la palabra secreta, se lo decimos al usuario, e incrementamos el número de errores en uno.

g) Terminamos el juego.
Cuando el programa llegue a este punto, el juego ha terminado. Si el número de errores ha llegado al máximo, el jugador pierde. En ese caso, imprimimos la figura final de la horca completa. En caso contrario, felicitamos al usuario. En ambos casos, le decimos al jugador cuál era la palabra secreta.

7. MANIPULACIÓN DE CADENAS.

El texto es una de las formas de datos más común en los programas. Ya sabemos cómo concatenar dos valores de tipo cadena mediante el operador `+`, pero podemos hacer muchas más cosas con las cadenas. Por ejemplo, podemos extraer cadenas parciales de un valor de tipo cadena, añadir y quitar espaciados, convertir letras de mayúsculas a minúsculas, y comprobar si las cadenas están formateadas de manera adecuada. Incluso podemos escribir programas en Python para acceder al **portapapeles** (clipboard) para copiar y pegar texto.[4] En este capítulo aprenderemos a hacer todo esto y mucho más.

7.1. TRABAJAR CON CADENAS.

En esta sección vamos a ver algunas de las formas con las que Python nos permite escribir, imprimir, y acceder a las cadenas en nuestros programas.

LITERALES DE CADENA (STRING LITERALS).

En Python, escribir valores de tipo cadena es bastante sencillo: siempre empiezan y terminan con una comilla simple. Pero entonces, ¿cómo podemos escribir una comilla dentro de una cadena? No podemos escribir `'That is Alice's cat.'`, porque Python pensará que la cadena termina justo después de `Alice`, y el resto de la cadena (`s cat.'`) será un código erróneo. Afortunadamente, hay otras formas de escribir cadenas.

Comillas dobles.

Las cadenas pueden empezar y terminar con comillas dobles, de la misma forma que empiezan y terminan con comillas simples. Una ventaja de utilizar las comillas dobles es que una cadena puede incluir un carácter de comilla simple dentro de ella. Escribe lo siguiente en el shell interactivo:

```
>>> spam = "That is Alice's cat"
```

Como la cadena empieza y termina con comillas dobles, Python sabe que la comilla simple es parte de la cadena, y que no marca el final de la cadena. Sin embargo, si nuestra cadena incluye comillas simples y comillas dobles dentro de ella, necesitaremos los caracteres de escape.

Caracteres de escape.

Un **carácter de escape** nos permite usar caracteres que de otra forma sería imposible incluir en una cadena. Un carácter de escape consiste en una barra inversa (`\`) (backslash) seguida del carácter que queremos añadir a la cadena. (A pesar de consistir realmente en dos caracteres, se le denomina comúnmente carácter de escape). Por ejemplo, el carácter de escape para la comilla simple es `\'`. Podemos usar esto dentro de una cadena que comience y termine con comilla simples para añadir una comilla simple a la cadena.

[4] Cuando en un ordenador copiamos (o cortamos) y pegamos (por ejemplo, cuando copiamos y pegamos texto en un programa tipo Word, o cuando copiamos y pegamos archivos de una carpeta a otra), el sistema operativo utiliza el **portapapeles** para almacenar temporalmente esa información. Cuando copiamos o cortamos, la información se lleva al portapapeles, y se recupera al pegar o mover esa información en el destino deseado. El portapapeles básico solo permite una copia, es decir, no permite almacenar múltiples archivos, imágenes, o textos, solo la última información copiada. Esa información se pierde al reiniciar el sistema.

Para ver cómo funcionan los caracteres de escape, escribe lo siguiente en el shell interactivo:

```
>>> spam = 'Say hi to Bob\'s mother.'
```

Python sabe que, como la comilla simple en `Bob\'s` está precedida de una barra inversa, no se trata de una comilla que indique el final de la cadena. Los caracteres de escape `\'` y `\"` nos permiten poner comillas simples y dobles dentro de nuestras cadenas.

La siguiente tabla lista los distintos caracteres de escape que podemos usar:

Escape Characters

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

A modo de ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

Cadenas en bruto.

Podemos poner una `r` delante de las comillas que marcan el principio de una cadena para convertirla en una *cadena en bruto*. Una **cadena en bruto** ignora por completo todos los caracteres de escape e imprime cualquier barra inversa que aparezca en la cadena. Por ejemplo:

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Como se trata de una cadena en bruto, Python considera que la barra inversa es parte de la cadena, y no el comienzo de un carácter de escape. Las cadenas en bruto son útiles cuando estamos escribiendo valores de tipo cadena que contienen muchas barras inversas, como las cadenas para las **expresiones regulares**, de las que hablaremos en capítulos posteriores.

Cadenas multilínea con comillas triples.

Aunque podemos usar el carácter de escape `\n` para añadir un salto de línea a una cadena, a menudo suele ser más fácil usar las **cadenas multilínea**. En Python, una cadena multilínea empieza y termina con tres comillas simples o con tres comillas dobles. Cualquier comilla, tabulación, o salto de línea entre las "comillas triples" se considerarán parte de la cadena. Las reglas de tabulación de Python para los bloques no aplican a las líneas que estén dentro de una cadena multilínea.

Abre el editor de archivos y escribe lo siguiente:

```
print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob''')
```

Guarda este programa como `catnapping.py` y ejecútalo. La salida se parecerá a lo siguiente:

```
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
```

Observar que el carácter de comilla simple en `Eve's` no tiene que escribirse como un carácter de escape. Los caracteres de escape para las comillas simples y dobles son opcionales en las cadenas multilínea. La siguiente llamada a la función `print()` debería imprimir exactamente el mismo texto, pero sin usar una cadena multilínea:

```
print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat
burglary, and extortion.\n\nSincerely,\nBob')
```

Comentarios multilínea.

Aunque el carácter de almohadilla (#) marca el comienzo de un comentario para el resto de la línea, a menudo suelen usarse las cadenas multilínea para marcar comentarios que se extienden a lo largo de múltiples líneas. El siguiente código es perfectamente válido en Python:

```
"""This is a test Python program.
Written by Alejandro Martinez (alextecnoeso.wordpress.com)

This program was designed for Python 3, not for Python 2.
"""

def spam():
"""This is a multiline comment to help
explain what the spam() function does."""
print('Hello!')
```

INDEXAR Y CORTAR CADENAS.

Las cadenas usan índices y cortes exactamente igual que las listas. Así, podemos imaginar que la cadena `'Hello world!'` es una lista, y que cada carácter en la cadena es un objeto con su índice correspondiente:

'	H	e	l	l	o		w	o	r	l	d	!	'
	0	1	2	3	4	5	6	7	8	9	10	11	

Los caracteres de espaciado y de exclamación se incluyen en el recuento de caracteres de la cadena, por lo que `'Hello world!'` es una cadena de 12 caracteres, desde `H` en el índice 0 hasta `!` en el índice 11.

Escribe lo siguiente en el shell interactivo:

```
>>> spam = 'Hello world!'
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
'Hello'
>>> spam[:5]
'Hello'
>>> spam[6:]
'world!'
```

Si especificamos un índice, obtendremos el carácter que está en esa posición en la cadena. Si especificamos un rango desde un índice hasta otro índice, el índice de comienzo estará incluido pero el índice de finalización no. Ésta es la razón por la que, si la cadena `spam` contiene `'Hello world!'`, la instrucción `spam[0:5]` devuelve `'Hello'`. La subcadena que obtenemos al escribir `spam[0:5]` incluye todos los caracteres desde `spam[0]` hasta `spam[4]`, dejando fuera el espacio en el índice 5.

Notar que el hecho de cortar una cadena no modifica la cadena original. Podemos capturar el corte de una variable tipo cadena en otra variable independiente. Escribe lo siguiente en el shell interactivo:

```
>>> spam = 'Hello world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

Al cortar y almacenar la subcadena resultante en una variable independiente, disponemos tanto de la cadena completa como de la subcadena, y podemos acceder a ambas fácilmente.

LOS OPERADORES IN Y NOT IN CON CADENAS.

Los operadores `in` y `not in` pueden usarse con las cadenas de la misma forma que los usamos con las listas. Una expresión con dos cadenas unidas por un `in` o un `not in` siempre se evaluará a un valor Booleano `True` o `False`. Escribe lo siguiente en el shell interactivo:

```
>>> 'Hello' in 'Hello world!'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello world!'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

Estas expresiones comprueban si la primera cadena (la cadena exacta, distinguiendo entre mayúsculas y minúsculas) está en la segunda cadena.

7.2. MÉTODOS DE CADENAS.

Existen varios métodos que permiten analizar cadenas o crear cadenas transformadas. En esta sección describiremos los métodos que usaremos de forma más habitual.

LOS MÉTODOS UPPER(), LOWER(), ISUPPER(), E ISLOWER().

Los métodos de cadena `upper()` y `lower()` devuelven una nueva cadena en la que todas las letras de la cadena original se han convertido en mayúsculas o minúsculas, respectivamente. Los caracteres que no sean de tipo letra permanecen inalterados. Escribe lo siguiente en el shell interactivo:

```
>>> spam = 'Hello world!'
>>> spam.upper()
'HELLO WORLD!'
>>> spam
'Hello world!'
>>> spam.lower()
'hello world!'
>>> spam
'Hello world!'
```

Observar que estos métodos no cambian la cadena en sí; en vez de eso, devuelven nuevas cadenas. Si queremos cambiar la cadena original, debemos llamar a los métodos `upper()` y `lower()` sobre la cadena, y entonces asignar la nueva cadena a la variable que almacenaba la cadena original. Por consiguiente, si queremos cambiar a mayúsculas la cadena almacenada en `spam`, deberíamos escribir `spam = spam.upper()`, en vez de escribir simplemente `spam.upper()`. (Esto es como si la variable `eggs` contuviese el valor 10. Si solo escribiésemos `eggs + 3` no cambiaríamos el valor de `eggs`; para cambiarlo, deberíamos escribir `eggs = eggs + 3`).

Los métodos `upper()` y `lower()` son útiles en aquellos casos en los que necesitamos hacer una comparación que no discrimine entre mayúsculas y minúsculas. Las cadenas `'great'` y `'GREat'` no son iguales. Pero en el siguiente programa no importa si el usuario ha tecleado `Great`, `GREAT`, o `grEAT`, porque la cadena se ha convertido a minúsculas:

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is good.')
```

Al ejecutar este programa la pregunta se muestra por pantalla, y si el usuario inserta una variación de `great`, como por ejemplo `GREat`, el programa proporciona como salida `I feel great too`. El hecho de poder añadir el código para manejar variaciones o errores en los datos de entrada de los usuarios (como mayúsculas o minúsculas inconsistentes) hará que nuestros programas sean más robustos y menos propensos a errores.

```
How are you?
GREat
I feel great too.
```

Los métodos `isupper()` e `islower()` devolverán un valor Booleano `True` si la cadena, que debe tener al menos una letra, tiene todas sus letras en mayúsculas o en minúsculas, respectivamente. En caso contrario, estos métodos devuelven `False`.

Escribe lo siguiente en el shell interactivo, y observa lo que devuelve cada llamada al método respectivo:

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

Como los métodos `upper()` y `lower()` devuelven cadenas, podemos llamar al resto de métodos de cadenas *sobre* las cadenas devueltas por estos métodos. Las expresiones para hacer esto tendrán la forma de una secuencia de llamadas a métodos. Escribe lo siguiente en el shell interactivo:

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

LOS MÉTODOS ISX.

Además de los métodos `isupper()` e `islower()`, hay otros muchos métodos de cadenas cuyos nombres comienzan con la palabra `is`. Estos métodos devuelven un valor Booleano que describe la naturaleza de la cadena. He aquí algunos métodos `isX` muy comunes:

- `isalpha()` devuelve `True` si la cadena consiste solo en letras, y no está vacía.
- `isalnum()` devuelve `True` si la cadena consiste solo en letras y números, y no está vacía.
- `isdecimal()` devuelve `True` si la cadena consiste solo en caracteres numéricos, y no está vacía.
- `isspace()` devuelve `True` si la cadena consiste solo en espacios, tabulaciones, y caracteres de nueva línea, y no está vacía.
- `istitle()` devuelve `True` si la cadena consiste solo en palabras que comienzan con una letra mayúscula seguida de letras solo en minúscula.

Escribe lo siguiente en el shell interactivo:

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isalnum()
True
```

```
>>> '123'.isdecimal()
True
>>> 'hello123'.isdecimal()
False
>>> '    '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

Los métodos `isX` son útiles cuando necesitamos validar datos de entrada de usuario. Por ejemplo, el siguiente programa le pide repetidamente al usuario su edad y una contraseña hasta que proporciona unos datos válidos. Abre una nueva ventana en el editor de archivos, escribe el programa, y guárdalo como `validateInput.py`:

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')
```

En el primer bucle `while` le preguntamos al usuario su edad, y almacenamos su respuesta en `age`. Si `age` es una valor (decimal) válido, salimos de este primer bucle `while` y nos movemos al segundo, que le pide una contraseña. En caso contrario, le informamos de que la edad que proporcione debe ser un número, y le pedimos que vuelva a insertar su edad. En el segundo bucle `while` le pedimos una contraseña, la almacenamos en `password`, y salimos del bucle si la contraseña escrita es alfanumérica. Si no lo es, le decimos al usuario que la contraseña ha de ser alfanumérica, y le pedimos que vuelva a escribirla. Al ejecutar el programa, la salida debe parecerse a la siguiente:

```
Enter your age:
forty two
Please enter a number for your age.
Enter your age:
42
Select a new password (letters and numbers only):
secr3t!
Passwords can only have letters and numbers.
Select a new password (letters and numbers only):
secr3t
>>>
```

Llamando a los métodos `isdecimal()` e `isalnum()` sobre variables seremos capaces de comprobar si los valores almacenados en ellas son o no son decimales, o si son o no son alfanuméricos. En este programa, estas comprobaciones nos han ayudado a rechazar la entrada `forty two` y a aceptar `42`, y también a rechazar `secr3t!` y a aceptar `secr3t`.

LOS MÉTODOS STARTSWITH() Y ENDSWITH().

Los métodos `startswith()` y `endswith()` devuelven `True` si la cadena sobre la que se llaman comienza o termina (respectivamente) con la cadena que le pasamos al método como argumento. En caso contrario, devuelven `False`. Escribe lo siguiente en el shell interactivo:

```
>>> 'Hello world!'.startswith('Hello')
True
>>> 'Hello world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello world!'.startswith('Hello world!')
True
>>> 'Hello world!'.endswith('Hello world!')
True
```

Estos métodos son alternativas útiles al operador de igualdad `==` en aquellos casos en los que solo necesitamos comprobar si la primera o la última parte de una cadena (y no la cadena completa) son iguales a otra cadena dada.

LOS MÉTODOS JOIN() Y SPLIT().

El método `join()` es útil cuando tenemos una lista de cadenas que deben combinarse en un único valor de tipo cadena. El método `join()` se llama sobre una cadena, se le pasa una lista de cadenas, y devuelve una cadena. La cadena que devuelve es la concatenación de todas las cadenas en la lista que le ha pasado como argumento. ¿Y cuál es el papel que desempeña la cadena sobre la que se llama al método? Para entenderlo, escribe lo siguiente en el shell interactivo:

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Notar que la cadena sobre la que se llama al método `join()` se inserta entre cada cadena de la lista que se le pasa como argumento. Por ejemplo, cuando llamamos a `join(['cats', 'rats', 'bats'])` sobre la cadena `', '`, la cadena devuelta es `'cats, rats, bats'`.

Debemos recordar que el método `join()` se llama sobre un valor de tipo cadena, y que se le pasa una lista de cadenas. Es fácil confundirse y hacerlo al revés. Por su parte, el método `split()` hace lo opuesto: se le llama sobre una cadena y devuelve una lista de cadenas. Escribe lo siguiente en el shell interactivo:

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

Por defecto, el método `split()` divide la cadena `'My name is Simon'` allá donde encuentre caracteres de espacio en blanco como el espacio, la tabulación, o el cambio de línea. Estos caracteres de espacio en blanco no se incluyen en los caracteres de las cadenas devueltas en la lista que retorna. Al método `split()` también podemos pasarle una cadena de delimitación para especificar una separación diferente a la separación por defecto.

Por ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

Una aplicación muy común de `split()` es la separación de una cadena multilínea mediante los caracteres de nueva línea. Escribe el siguiente código en el shell interactivo:

```
>>> spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment".

Please do not drink it.
Sincerely,
Bob'''
>>> spam.split('\n')
['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the
fridge', 'that is labeled "Milk Experiment".', '', 'Please do not drink it.',
'Sincerely,', 'Bob']
```

Pasándole a `split()` el argumento `'\n'` podemos dividir la cadena multilínea almacenada en `spam` en las distintas líneas que contiene, y devolver una lista en la que cada elemento se corresponde con una de las líneas de la cadena.

ALINEAR TEXTO CON LOS MÉTODOS RJUST(), LJUST(), Y CENTER().

Los métodos de cadena `rjust()` y `ljust()` devuelven una versión formateada de la cadena sobre la que se llaman con espacios insertados para alinear (justify) el texto. El primer argumento de ambos métodos es un entero para especificar la longitud de la cadena alineada. Escribe lo siguiente en el shell interactivo:

```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello World'.rjust(20)
'         Hello World'
>>> 'Hello'.ljust(10)
'Hello     '
```

`'Hello'.rjust(10)` indica que queremos alinear `'Hello'` a la derecha en una cadena de longitud 10. Como `'Hello'` tiene cinco caracteres, se añadirán cinco espacios a su izquierda, dando lugar a una cadena de 10 caracteres con `'Hello'` alineada a la derecha.

Una técnica muy habitual con estas funciones es pasarles como argumento la suma de la longitud de la cadena más el espaciado extra deseado para la anchura total de la cadena de salida:

```
>>> str = 'Hello'
>>> extra_width =  5
>>> str.rjust(len(str) + extra_width)
'     Hello'
```

Este código alinea la cadena `'Hello'` a la derecha añadiendo 5 espacios a la izquierda, para dar lugar a una cadena con una longitud total de 10 caracteres. Esta técnica es muy útil cuando desconocemos la longitud de la cadena que queremos alinear.

El segundo argumento opcional de los métodos `rjust()` y `ljust()` sirve para especificar otro carácter de relleno distinto de los espacios en blanco. Escribe lo siguiente en el shell interactivo:

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

Por otra parte, el método `center()` funciona igual que `rjust()` y `ljust()`, pero en su caso centra el texto en lugar de alinearlo. Escribe lo siguiente en el shell interactivo:

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

Estos métodos son especialmente útiles cuando necesitamos imprimir datos tabulares (esto es, en forma de tabla) con el espaciado adecuado. Abre una nueva ventana en el editor de archivos, escribe el siguiente código, y guárdalo como `picnicTable.py`:

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

En este programa definimos una función `printPicnic()` que recibirá un diccionario como argumento, y usará los métodos `center()`, `ljust()`, y `rjust()` para mostrar la información presente en el diccionario en la forma de una tabla correctamente alineada.

El diccionario que le pasamos a `printPicnic()` es `picnicItems`. En `picnicItems` tenemos 4 sandwiches, 12 manzanas, 4 tazas, y 8000 galletas. Queremos organizar esta información en dos columnas, con el nombre del elemento a la izquierda y la cantidad a la derecha.

Para hacerlo, decidimos cómo de anchas queremos que sean las columnas izquierda y derecha. Es por ello que a la función `printPicnic()` también le pasamos como argumentos las anchuras deseadas para estas columnas. `printPicnic()` recibe un diccionario, el valor `leftWidth` para la anchura de la columna izquierda de la tabla, y el valor `rightWidth` para la anchura de la columna derecha. Esta función imprime un título, `PICNIC ITEMS`, centrado sobre la tabla. A continuación, itera a través del diccionario, imprimiendo cada par clave – valor en una línea con la clave alineada a la izquierda (y completada con puntos), y el valor alineado a la derecha (y completado con espacios).

Después de definir la función `printPicnic()`, definimos el diccionario `picnicItems` y llamamos a la función dos veces, pasándole diferentes anchuras para las columnas de la izquierda y de la derecha.

Al ejecutar este programa, los elementos del diccionario se muestran por pantalla dos veces. La primera vez la columna de la izquierda tiene una anchura de 12 caracteres, y la columna de la derecha de 5 caracteres. La segunda vez tienen 20 y 6 caracteres de ancho, respectivamente (ver figura):

```
---PICNIC ITEMS--
sandwiches..     4
apples......    12
cups........     4
cookies..... 8000
```

```
-------PICNIC ITEMS-------
sandwiches.....      4
apples.....         12
cups.....            4
cookies.....      8000
```

El uso de los métodos `center()`, `rjust()`, y `ljust()` nos permite asegurar que las cadenas están correctamente alineadas, incluso cuando no estamos seguros de cuántos caracteres tienen estas cadenas.

QUITAR ESPACIOS EN BLANCO CON STRIP(), RSTRIP(), Y LSTRIP().

En ocasiones necesitaremos eliminar caracteres de espacios en blanco (espaciados, tabulaciones, y saltos de línea) del lado izquierdo, derecho, o a ambos lados de una cadena. El método `strip()` devuelve una nueva cadena sin ningún carácter de espacio en blanco al principio o al final de la misma. Los métodos `lstrip()` y `rstrip()` quitan los caracteres de espacio en blanco de los extremos izquierdo y derecho de una cadena, respectivamente. Escribe lo siguiente en el shell interactivo:

```
>>> spam = '   Hello World   '
>>> spam.strip()
'Hello World'
>>> spam.lstrip()
'Hello World   '
>>> spam.rstrip()
'   Hello World'
```

Opcionalmente también podemos pasarles un argumento tipo cadena para especificar qué caracteres deben eliminarse. Escribe lo siguiente en el shell interactivo:

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

Pasándole a `strip()` el argumento `'ampS'` le estamos diciendo que elimine las ocurrencias de los caracteres `a`, `m`, `p`, y `S` mayúscula de ambos extremos de la cadena almacenada en `spam`. El orden de los caracteres a eliminar no es relevante: `strip('ampS')` hará lo mismo que `strip('Spam')` o que `strip('Sapm')`.

COPIAR Y PEGAR CADENAS CON EL MÓDULO PYPERCLIP.

El módulo `pyperclip` incorpora las funciones `copy()` y `paste()`, que nos permiten enviar y recibir texto del portapapeles (clipboard) del ordenador. El hecho de poder enviar la salida de nuestro programa al portapapeles nos facilitará poder pegarla en un correo electrónico, en un procesador de textos, o en cualquier otro software.

El módulo `pyperclip` no viene por defecto en Python, y hay que instalarlo. Para hacerlo, sigue las instrucciones indicadas en el anexo A de este texto para instalar módulos de terceros. Después de instalar el módulo `pyperclip`, escribe lo siguiente en el shell intreractivo:

```
>>> import pyperclip
>>> pyperclip.copy('Hello world!')
>>> pyperclip.paste()
'Hello world!'
```

Por supuesto, si algo externo a nuestro programa modifica los contenidos del portapapeles, la función `paste()` devolverá esos contenidos.

Por ejemplo, si copiamos esta frase al portapapeles y a continuación llamamos a `paste()`, obtendremos lo siguiente:

```
>>> pyperclip.paste()
'Por ejemplo, si copiamos esta frase al portapapeles y a continuación
llamamos a paste(), obtendremos lo siguiente:'
```

Ejecutar programas de Python fuera del IDLE.

Hasta ahora hemos ejecutado nuestros programas de Python usando el shell interactivo y el editor de archivos del IDLE. Sin embargo, habitualmente no querremos pasar por el proceso de abrir el IDLE y el archivo .py con el código de Python cada vez que vayamos a ejecutar un programa.

Afortunadamente, hay algunos atajos para hacer que la ejecución de programas Python sea más sencilla. Los pasos a dar son ligeramente distintos para Windows, OS X, y Linux. El anexo B de este texto explica cómo podemos ejecutar programas Python (en Windows) y cómo pasarles argumentos a través de la ventana de comandos. Esto es muy interesante, ya que no podemos pasar argumentos a nuestros programas usando el IDLE.

7.3. INTERPOLACIÓN DE CADENAS.

Normalmente, si queremos usar dentro de una cadena otros valores de tipo cadena que están almacenados en variables, debemos usar el operador de concatenación `+`. Por ejemplo:

```
>>> name = 'Alice'
>>> event = 'party'
>>> location = 'the pool'
>>> day = 'Saturday'
>>> time = '6:00pm'
>>> print('Hello, ' + name + '. Will you go to the ' + event
          + ' at ' + location + ' this ' + day + ' at ' + time + '?')
Hello, Alice. Will you go to the party at the pool this Saturday at 6:00pm?
```

Como vemos, es muy engorroso tener que escribir una línea que concatene varias cadenas. En vez de eso, podemos usar una técnica llamada **interpolación de cadenas**, que nos permite poner **marcadores** como `%s` dentro de una cadena. Estos marcadores se denominan **especificadores de conversión**. Una vez añadidos los especificadores de conversión, podemos poner todos los nombres de las variables al final de la cadena. Cada `%s` se reemplazará con una de las variables al final de la línea, en el orden en el que escribimos las variables. Por ejemplo, el siguiente código hace exactamente lo mismo que el código anterior:

```
>>> name = 'Alice'
>>> event = 'party'
>>> location = 'the pool'
>>> day = 'Saturday'
>>> time = '6:00pm'
>>> print('Hello, %s. Will you go to the %s at %s this %s at %s?' % (name,
event, location, day, time))
Hello, Alice. Will you go to the party at the pool this Saturday at 6:00pm?
```

Notar que el primer nombre de variable se usa en lugar del primer `%s`, la segunda variable en lugar del segundo `%s`, etc. Debemos tener el mismo número de especificadores `%s` que de variables.

Otra ventaja de usar interpolación de cadenas en lugar de la concatenación de cadenas es que la interpolación funciona con cualquier tipo de datos, y no solo con cadenas. Todos los valores se convierten automáticamente en cadenas.

Por ejemplo, si concatenamos un entero con una cadena, obtenemos un mensaje de error:

```
>>> spam = 42
>>> print('Spam == ' + spam)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    print('Spam == ' + spam)
TypeError: can only concatenate str (not "int") to str
```

En efecto, la concatenación de cadenas solo puede combinar dos cadenas, pero `spam` es un entero. Para no obtener un mensaje de error, deberíamos haber recordado escribir `str(spam)` en vez de `spam`. Sin embargo, escribe lo siguiente en el shell interactivo:

```
>>> spam = 42
>>> print('Spam is %s' % (spam))
Spam is 42
```

Otra gran ventaja de la interpolación es que no es obligatorio reemplazar especificadores de conversión por variables. También podemos reemplazarlos por expresiones que involucren a las variables listadas, como en el siguiente ejemplo:

```
>>> num1 = 12
>>> num2 = 34
>>> print('The addition of %s and %s is %s' % (num1, num2, num1 + num2))
The addition of 12 and 34 is 46
```

7.4. PROYECTO: GESTOR DE CONTRASEÑAS.

Todos tenemos cuentas en muchas webs distintas. Una muy mala costumbre es usar la misma contraseña para todas ellas, porque si una de estas webs tiene una brecha de seguridad, los hackers obtendrán la contraseña de todas nuestras cuentas. Lo mejor es usar un software de gestión de contraseñas que utilice una contraseña maestra para desbloquear el gestor de contraseñas. A continuación, puedes copiar al portapapeles la contraseña específica de una determinada cuenta, y pegarla en la ventana de acceso a la web en cuestión. El programa gestor de contraseñas que desarrollaremos en este ejemplo no es seguro, pero constituye una muestra de cómo funcionan este tipo de programas.

PASO 1: DISEÑO DEL PROGRAMA Y ESTRUCTURAS DE DATOS.

En este caso querremos ser capaces de ejecutar este programa desde la ventana de comandos, pasándole un argumento que será el nombre de la cuenta (por ejemplo, *email* o *blog*). La contraseña de esa cuenta se copiará en el portapapeles, de forma que el usuario pueda pegarla en la ventana de acceso a la web. De esta forma, el usuario puede tener contraseñas largas y complicadas sin necesidad de memorizarlas.

Abre una nueva ventana del editor de archivos y guarda el programa (vacío) como `pw.py`. Para poder ejecutar un programa desde la ventana de comandos, debemos comenzar el programa con una línea `#!` (línea shebang, ver anexo B). También vamos a escribir con comentarios una breve descripción del programa. Como queremos asociar cada nombre de cuenta con su contraseña correspondiente, podemos almacenarlo todo como cadenas dentro de un diccionario. El diccionario será la estructura de datos que organice nuestros datos de cuentas y contraseñas.

Escribe un programa que se parezca a lo siguiente:

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
'luggage': '12345'}
```

PASO 2: MANEJO DE LOS ARGUMENTOS PASADOS MEDIANTE LA VENTANA DE COMANDOS.

Cuando ejecutamos un programa desde la ventana de comandos, en ella escribimos el nombre del programa seguido de los argumentos que necesita. Toda esta información se almacena en una variable tipo lista llamada `sys.argv`. El primer elemento de la lista `sys.argv` siempre es una cadena que contiene el nombre de archivo del programa (`'pw.py'`), el segundo elemento es el primer argumento de ventana de comandos, el tercer elemento es el segundo argumento de ventana de comandos, etc. Para nuestro programa, el único argumento que pasamos es el nombre de la cuenta cuya contraseña queremos recuperar. Desde el programa podemos obtener y guardar este nombre de cuenta accediendo al segundo elemento (índice 1) de la lista `sys.argv`, mediante la instrucción `account = sys.argv[1]`. Como este **argumento de ventana de comandos** es obligatorio, hemos de mostrarle al usuario un mensaje de información en caso de que se olvide de pasarlo (esto es, si la lista `sys.argv` contiene menos de dos valores). Completa el programa como mostramos:

```
import sys, pyperclip
if len(sys.argv) < 2:
    print('Usage in command window: pw.py account_name')
    sys.exit()

account = sys.argv[1] # first command line arg is the account name
```

PASO 3: COPIAR LA CONTRASEÑA CORRECTA.

Ahora que hemos guardado el nombre de la cuenta como una cadena en la variable `account`, debemos comprobar si ésta existe como clave en el diccionario `PASSWORDS`. Si es así, queremos copiar el valor de la clave en el portapapeles usando `pyperclip.copy()`. (Como vamos a usar el módulo `pyperclip`, debemos importarlo). Notar que en realidad no necesitamos la variable `account`; en vez de eso, podríamos usar simplemente `sys.argv[1]`. Sin embargo, usar una variable llamada `account` hace que el programa sea mucho más legible. Completa tu programa con el siguiente código:

```
if account in PASSWORDS:
    pyperclip.copy(PASSWORDS[account])
    print('Password for ' + account + ' copied to clipboard.')
else:
    print('There is no account named ' + account)
```

Este nuevo código busca en el diccionario `PASSWORDS` el nombre de la cuenta indicado. Si esa cuenta existe como clave del diccionario, obtenemos su valor asociado, lo copiamos al portapapeles, e imprimimos un mensaje diciendo que hemos copiado el valor. En caso contrario, imprimimos un mensaje que diga que no hay una cuenta con ese nombre.

Con esto hemos terminado el programa. Usando las instrucciones del anexo B para lanzar programas desde la ventana de comandos, ahora ya disponemos de una forma rápida de copiar nuestras contraseñas al

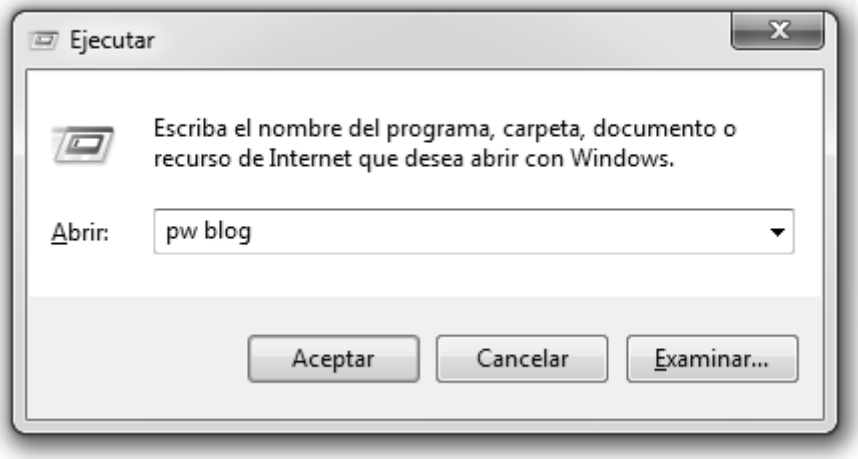
portapapeles. Por supuesto, será necesario editar el diccionario `PASSWORDS` siempre que queramos modificar o añadir una cuenta.

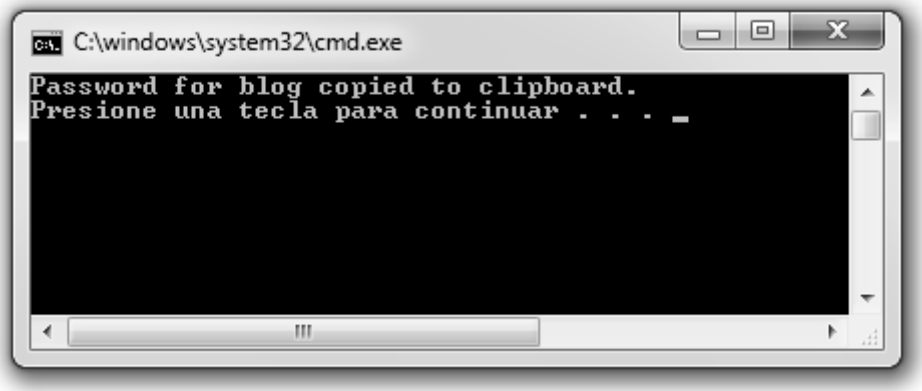
Evidentemente, es probable que no queramos guardar todas nuestras contraseñas en un lugar donde cualquiera pueda acceder y copiarlas. Pero a pesar de este (grave) inconveniente, podemos modificar este programa y usarlo para copiar un texto cualquiera al portapapeles. Por ejemplo, digamos que queremos enviar múltiples correos que tienen muchos párrafos que son idénticos. Podemos poner cada párrafo como un valor en el diccionario `PASSWORD` (llegados a este punto, probablemente querremos cambiarle el nombre al diccionario), para así disponer de una forma rápida de seleccionar y copiar en el portapapeles uno de los muchos párrafos que todos los correos tienen en común.

Para poder ejecutar un programa desde la ventana de comandos de Windows (tecla de Windows + R), hemos de crear un archivo por lotes (archivo batch). (Para más detalles, ver anexo B). Escribe lo siguiente en el editor de archivos, y guarda el archivo como `pw.bat` en la misma carpeta en la que se localice el archivo `pw.py`.

```
@py.exe C:\Users\Usuario\Dropbox\PYTHON\PROGRAMAS\pw.py %*
@pause
```

(La ruta especificada dependerá de dónde tengas guardado el archivo `pw.py`. Esa ruta deberá haber sido añadida a la variable de entorno PATH de Windows, ver anexo B). Habiendo creado este archivo batch, para ejecutar el programa simplemente tenemos que abrir la ventana de diálogo "ejecutar" (WIN + R) y escribir `pw account name`.





Si ahora abrimos un editor de textos y pegamos (CTRL + V), recuperaremos del portapapeles la contraseña asociada a la cuenta `blog`:

`VmALvQyKAxiVH5G8v01if1MLZF3sdt`

7.5. PROYECTO: PONER VIÑETAS A LA LISTA DE UNA WIKI.

Al editar un artículo de Wikipedia, podemos crear una lista con viñetas poniendo cada elemento de la lista en su propia línea y ubicando un asterisco justo delante. Pero imaginar que tenemos una lista realmente larga, y que queremos añadir una viñeta a cada uno de los elementos de la lista. Añadir a mano un asterisco delante de cada una de las líneas sería un engorro. Afortunadamente, podemos automatizar esta tarea con un pequeño programa de Python.

El programa `bulletPointAdder.py` obtendrá el texto del portapapeles (donde previamente lo habremos copiado), añadirá un asterisco al principio de cada línea, y a continuación pegará este nuevo texto de vuelta al portapapeles. Por ejemplo, si copiáramos el siguiente texto al portapapeles:

```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

, y a continuación ejecutásemos el programa `bulletPointAdder.py`, el portapapeles pasaría a contener lo siguiente:

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

Este texto está listo para ser pegado en el artículo de Wikipedia que estamos editando, en la forma de una lista con viñetas.

PASO 1: COPIAR Y PEGAR DESDE EL PORTAPAPELES.

Queremos que el programa `bulletPointAdder.py` haga lo siguiente:

1) Pegar texto del portapapeles.
2) Hacer algo con ese texto.
3) Copiar el nuevo texto al portapapeles.

El segundo paso es un poco más complicado. Afortunadamente, los pasos 1 y 3 son bastante sencillos: únicamente implican usar las funciones `pyperclip.copy()` y `pyperclip.paste()`. Por ahora solo vamos a escribir la parte del programa que se encarga de efectuar las tareas 1 y 3. Escribe el siguiente código, y guarda el programa como `bulletPointAdder.py`:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# STILL TO DO: Separate lines and add stars.

pyperclip.copy(text)
```

El comentario `STILL TO DO` es un recordatorio de que todavía debemos completar esta parte del programa. Nuestro siguiente paso es, precisamente, implementar ese trozo de código.

PASO 2: SEPARAR LAS LÍNEAS DEL TEXTO Y AÑADIR LOS ASTERISCOS.

La llamada a `pyperclip.paste()` devuelve todo el texto contenido en el portapapeles en la forma de una gran cadena. Utilizando el ejemplo previo, la cadena almacenada en `text` se parecería a lo siguiente:

```
'Lists of animals\nLists of aquarium life\nLists of biologists by author abbreviation\nLists of cultivars'
```

Los caracteres `\n` de nueva línea en esta cadena hacen que la cadena se muestre en varias líneas cuando se imprime o se pega desde el portapapeles. Hay muchas "líneas" en este valor tipo cadena. Y nosotros queremos añadir un asterisco al principio de cada una de ellas.

Podríamos escribir un código que busque cada carácter `\n` de nueva línea en la cadena, y a continuación, que añada el asterisco justo después de eso. Pero es mucho más sencillo usar el método `split()` para devolver una lista de cadenas (con una cadena para cada línea de la cadena original), y después añadir el asterisco delante de cada cadena en la lista.

Escribe un programa que se parezca a lo siguiente:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)): # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list

pyperclip.copy(text)
```

Dividimos el texto allá donde haya un carácter de nueva línea para obtener una lista en la que cada elemento es una línea del texto original. Almacenamos la lista en `lines` y entonces iteramos a través de los elementos en `lines`. Para cada línea, añadimos un asterisco y un espacio al principio de la línea. Ahora, cada cadena en `lines` comienza con un asterisco.

PASO 3: UNIR LAS LÍNEAS MODIFICADAS.

La lista `lines` contiene ahora las líneas modificadas que comienzan con asteriscos. Pero `pyperclip.copy()` espera recibir un único valor tipo cadena, y no una lista de cadenas. Para construir este valor único de tipo cadena, debemos pasarle la lista `lines` al método `join()`. Completa el programa como se muestra:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()
```

```
# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)): # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list

text = '\n'.join(lines)
pyperclip.copy(text)
```

Al ejecutarse, el programa reemplaza el texto en el portapapeles con un texto que tiene asteriscos al principio de cada línea. El programa está ahora completo, y podemos probar a ejecutarlo con algún texto copiado en el portapapeles.

Incluso aunque no necesitemos automatizar esta tarea en concreto, tal vez podríamos necesitar algún otro tipo de manipulación de texto, como eliminar los espacios sobrantes al final de las líneas, o convertir texto de mayúsculas a minúsculas. Para cualquier cosa que necesitemos, siempre podemos usar el portapapeles como entrada y como salida.

7.6. EJERCICIOS DEL CAPÍTULO 7.

Ejercicio 7.1. Escribe una función que combine los dos primeros y los dos últimos caracteres de una cadena. Si la cadena contiene menos de dos caracteres, la función devolverá una cadena vacía. Guarda el programa como `Ejer7.1.py`.

Ejercicio 7.2. Escribe una función `changeChar(str, ch)` que reciba como argumentos una cadena de texto, y un carácter. La función debe reemplazar cada ocurrencia de ese carácter por un símbolo `$`. Por ejemplo, si le pasamos la cadena `'Susana'` y el carácter `'a'`, la función debería devolver `'Sus$n$'`. Guarda el programa como `Ejer7.2.py`.

Ejercicio 7.3. Escribe una función que permita añadir `'ing'` al final de una cierta cadena si la longitud de la cadena es, al menos, de 3. Si la cadena ya termina en `'ing'`, entonces la terminamos con `'ly'`. Si la longitud de la cadena es menor que 3, la deja inalterada. Guarda el programa como `Ejer7.3.py`.

Ejercicio 7.4. Escribe una función que reciba una lista de cadenas. La función debe devolver la longitud de la cadena más larga en la lista. Por ejemplo, si la lista es `['apples', 'oranges', 'cherries', 'banana']`, la función debería devolver 8. Guarda el programa como `Ejer7.4.py`.

Ejercicio 7.5. Escribe una función que reciba una cadena que contiene una frase. La función debe crear un diccionario donde la clave sea cada una de las palabras de la frase, y el valor asociado sea el número de ocurrencias de esa palabra en la frase. Prueba tu programa con la frase `'the quick brown dog jumps over the small black dog'`. Guarda el programa como `Ejer7.5.py`.

Ejercicio 7.6. Escribe una función que reciba una cadena e invierta el orden de todos sus caracteres. Por ejemplo, si la función recibe la cadena `'apples'`, debe devolver la cadena `'selppa'`. Guarda el programa como `Ejer7.6.py`.

Ejercicio 7.7. Escribe una función que reciba una cadena con una frase, y que invierta las palabras presentes en esa frase. Por ejemplo, si la función recibe la cadena `'the quick brown dog jumps over the small black dog'`, debe devolver `'dog black small the over jumps dog Brown quick the'`. Guarda el programa como `Ejer7.7.py`.

Ejercicio 7.8. Escribe una función que decida si una cadena es o no es un palíndromo. (Un palíndromo es una palabra o expresión que es igual si se lee de izquierdas a derechas que de derechas a izquierdas). Comprueba su funcionamiento con la cadena `'aibohphobia'`. El programa debe funcionar

independientemente de si los caracteres están en mayúsculas o minúsculas (esto es, la palabra `'aibOHphobia'` será detectada como palíndromo). Guarda el programa como `Ejer7.8.py`.

Ejercicio 7.9. Escribe un programa que elimine todos los caracteres de puntuación de una cadena. Para ello, define una cadena o una lista que incluya todos los caracteres de puntuación que deseamos eliminar de la cadena original, por ejemplo, `punctuations = '''!()-[]{};:'"\,<>./?@#$%^&*_~'''`. Prueba el programa con la cadena `my_str = "Hello!!!, he said ---and went."`. Guarda el programa como `Ejer7.9.py`.

Ejercicio 7.10. Escribe una función que cuente el número de ocurrencias de cada vocal en una cadena. Se recomienda usar un diccionario, donde las claves sean las vocales, y su valor el número de ocurrencias. Guarda el programa como `Ejer7.10.py`.

Ejercicio 7.11. ADN.
Escribe una función `pairWiseDifferences(sequence1, sequence2)` que compute la proporción de bases en la que difieren dos secuencias de ADN de la *misma longitud*. Los argumentos `sequence1` y `sequence2` son cadenas. Por ejemplo, si usamos `pairWiseDifferences('AGTC', 'AGTT')`, la función debería devolver 0,25. Guarda el programa como `Ejer7.11.py`.

Ejercicio 7.12. "Pig Latin".
Pig Latin es un lenguaje sin sentido. Para traducir una palabra de inglés a "pig latin" debemos mover la primera letra al final de la palabra y añadir "ay" al final. Por ejemplo, `'monkey'` se convierte en `'onkeymay'`, y `'word'` se convierte en `'ordway'`. Escribe un programa que reciba como entrada una cadena con una *frase*, para convertirla a "pig latin".

Cuidado: La frase estará compuesta por varias palabras separadas por espacios en blanco. Primero obtén todas las palabras de la frase, y después tradúcelas una a una a "pig latin". Guarda el programa como `Ejer7.12.py`.

Ejercicio 7.13. ROT13 es una técnica de encriptación que implica "rotar" cada letra de una palabra 13 posiciones. Rotar una letra significa moverla a lo largo del alfabeto, empezando de nuevo desde el principio si es necesario. Por ejemplo, si la letra es A y la rotamos 3 posiciones, se convierte en D. Si la letra es X y la rotamos 5 posiciones, se convierte en C.

Escribe una función llamada `rotateWord` que reciba como argumentos una cadena y un entero, y que devuelva una nueva cadena que contenga las letras de la original "rotadas" tantas posiciones como indica el entero pasado. Por ejemplo, `'cheer'` rotada 7 posiciones devolvería `'jolly'`, y `'melon'` rotada −10 es `'cubed'`.

PISTA: Puede que te interese usar las funciones integradas `ord`, que convierte un carácter a su código numérico, y `chr`, que convierte códigos numéricos a caracteres.

```
ord: 32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47
chr:      !   "   #   $   %   &   '   (   )   *   +   ,   -   .   /
ord: 48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63
chr:  0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?
ord: 64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79
chr:  @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
ord: 80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
chr:  P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _
ord: 96  97  98  99  100 101 102 103 104 105 106 107 108 109 110 111
chr:  `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o
ord: 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
chr:  p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~
```

Ejercicio 7.14. Impresión de tablas.
Escribe una función llamada `printTable()` que reciba una lista de listas de cadenas, y muestre una tabla bien organizada y formateada donde las columnas estén alineadas a la derecha. Asume que todas las listas internas contendrán el mismo número de cadenas.

Por ejemplo, la lista podría ser:

```
tableData = [['apples', 'oranges', 'cherries', 'banana'],
['Alice', 'Bob', 'Carol', 'David'],
['dogs', 'cats', 'moose', 'goose']]
```

La función `printTable()` debería imprimir lo siguiente:

```
  apples Alice  dogs
 oranges   Bob  cats
cherries Carol moose
  banana David goose
```

Pista: En primer lugar, tu código tendrá que encontrar la cadena más larga en las distintas listas internas, de forma que la columna sea lo suficientemente ancha como para que quepan todas las cadenas. Puedes guardar la anchura máxima de cada columna como una lista de enteros. La función `printTable()` puede empezar con `colWidths = [0] * len(tableData)`, lo cual creará una lista que contenga el mismo número de valores `0` que el número de listas internas en `tableData`. De esta forma, `colWidths[0]` puede almacenar la anchura de la cadena más larga en `tableData[0]`, `colWidths[1]` puede almacenar la anchura de la cadena más larga en `tableData[1]`, y así sucesivamente. A continuación, puedes hallar el valor más grande en la lista `colWidths` para obtener qué anchura entera pasarle al método de cadena `rjust()`.

Ejercicio 7.15. Tabla de multiplicar.
Escribe una tabla de multiplicar ordenada y formateada como la siguiente:

```
      1   2   3   4   5   6   7   8   9  10  11  12
   :-----------------------------------------------
 1:    1   2   3   4   5   6   7   8   9  10  11  12
 2:    2   4   6   8  10  12  14  16  18  20  22  24
 3:    3   6   9  12  15  18  21  24  27  30  33  36
 4:    4   8  12  16  20  24  28  32  36  40  44  48
 5:    5  10  15  20  25  30  35  40  45  50  55  60
 6:    6  12  18  24  30  36  42  48  54  60  66  72
 7:    7  14  21  28  35  42  49  56  63  70  77  84
 8:    8  16  24  32  40  48  56  64  72  80  88  96
 9:    9  18  27  36  45  54  63  72  81  90  99 108
10:   10  20  30  40  50  60  70  80  90 100 110 120
11:   11  22  33  44  55  66  77  88  99 110 121 132
12:   12  24  36  48  60  72  84  96 108 120 132 144
```

Ejercicio 7.16. Hundir la flota.
En este ejercicio construirás una versión simplificada para un jugador del juego clásico de hundir la flota. Aquí tendremos un solo barco oculto en una localización aleatoria en una cuadrícula de 5×5 casillas. El jugador dispone de 10 intentos para intentar hundir el barco. Para conseguirlo, te recomendamos seguir los siguientes pasos:

- Crea una variable `board` (tablero) que almacene una lista vacía.
- Crea una cuadrícula 5×5 inicializada toda a `'O'` y guárdala en `board`. Para ello:
  - Usa `range()` para construir un bucle que itere 5 veces.
  - Dentro del bucle, usa `append()` para adjuntar una lista con 5 `'O'` a `board`.

- Aprovecha el hecho de que `board` es una lista de listas: Escribe una función `printBoard` para mostrar los contenidos de `board`. Para ello, usa un bucle que itere a lo largo de los `5` elementos de la lista externa (cada uno de los cuales es una lista con cinco `'O'`, que se corresponde con una fila del tablero) y que los imprima por pantalla.
- A continuación, elimina las comillas de las `'O'`. Vuelve a imprimir el tablero por pantalla.
- Ahora, vamos a esconder el barco en una localización aleatoria. Como tienes una lista bidimensional, usa dos variables para almacenar la localización del barco, `shipRow` y `shipCol`. Define dos funciones, `randomRow` y `randomCol` que reciban el tablero como entrada. Estas funciones deberían devolver una fila y una columna aleatorias en el tablero, respectivamente. Usa `randint(0, len(board) - 1)`.
- Modifica la lista `board` para cambiar la `'O'` por una `'B'` en la posición del barco. No imprimas el tablero ahora: le estarías diciendo al jugador dónde está el barco.
- Por supuesto, el juego debe preguntarle al usuario que adivine la fila y la columna donde está escondido el barco. Almacena sus respuestas en las variables `guessRow` y `guessCol`.
- Le damos al usuario un máximo de `10` intentos para adivinar la posición del barco. Como tenemos la posición real del barco en el tablero y la conjetura actual del usuario, comprobamos la respuesta. Si el usuario acierta, mostramos un mensaje de felicitación, y el juego termina. Si el usuario falla, le mostramos un mensaje de error, y modificamos la lista `board` para cambiar la `'O'` por una `'X'` en las coordenadas proporcionadas por el usuario. Si el usuario falla los `10` intentos, proporcionamos un mensaje de error, imprimimos por pantalla el tablero (con una `'B'` en lugar de una `'O'` en la posición del barco), y el juego termina.

AMPLIACIÓN:

- El jugador podría insertar una fila o columna erróneas. Controla este hecho, y proporciona el mensaje de información pertinente, tanto al principio del juego, como cuando el jugador introduzca un dato erróneo.
- El jugador podría proporcionar dos veces la misma localización. Controla este hecho y muestra un mensaje de información cuando eso ocurra. Dale otra oportunidad y no descuentes este intento repetido del número restante de intentos.

Ejercicio 7.17. Escribe una función que implemente un código de sustitución. En el código de sustitución se sustituye una letra por otra. Un ejemplo sería A → Q, B → T, C → G, etc. La función debería recibir dos parámetros: una cadena con el mensaje a encriptar, y una cadena (o lista, o diccionario) con el mapeado de las 26 letras del alfabeto. La función debería devolver una cadena con la versión encriptada del mensaje original. Guarda el programa como `Ejer7.17.py`.

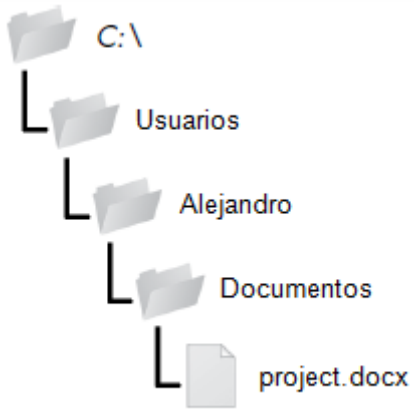
Ejercicio 7.18. Escribe una función que descodifique el mensaje encriptado del ejercicio previo. Esta función también debería recibir dos parámetros: una cadena con el mensaje encriptado, y una cadena (o lista, o diccionario) con el mapeado de las 26 letras del alfabeto. Esta función debería devolver una cadena con el mensaje decodificado original. Guarda el programa como `Ejer7.18.py`.

8. LECTURA Y ESCRITURA DE ARCHIVOS.

Las variables son un buen recurso para almacenar y acceder a la información durante la ejecuación de un un programa. Pero si queremos que los datos persistan incluso después de que nuestro programa haya terminado, debemos guardarlos en un archivo. En este capítulo aprenderemos la forma en la que Python permite crear, leer, y guardar archivos en el disco duro para lograr este almacenamiento permanente de datos.

8.1. ARCHIVOS Y RUTAS DE ARCHIVOS.

Todo archivo se caracteriza por dos propiedades básicas: un **nombre de archivo** (habitualmente escrito con una sola palabra) y una **ruta** ("path", en inglés). La ruta especifica la localización del archivo en el ordenador. Por ejemplo, en mi ordenador con Windows 7 podría tener un archivo de Word llamado *project.docx* en la ruta *C:\Usuarios\Alejandro\Documentos*. La parte del nombre de archivo después del punto se denomina **extensión** del archivo, y nos informa sobre el tipo de archivo (por ejemplo, *.docx* es la extensión de los archivos de Word, *.xlsx* es la de los archivos Excel, *.jpg* es una de las extensiones típicas de los archivos de imagen, etc.). Por su parte, las palabras *Usuarios*, *Alejandro*, y *Documentos* en la ruta del archivo indican **carpetas** (también llamados **directorios**). Las carpetas pueden contener archivos u otras carpetas. Así, el archivo *project.docx* está dentro de la carpeta *Documentos*, que a su vez está dentro de la carpeta *Alejandro*, que a su vez está dentro de la carpeta *Usuarios*. La figura muestra esta ornanización en forma de carpetas.



La parte C:\ de la ruta se denomina **carpeta raíz**, y es la que contiene a todas los demás carpetas. En Windows, a la carpeta raíz se la denomina **C:**, o también **unidad C:**. En OS X y Linux, la carpeta raíz es /. En estos apuntes siempre usaremos la notación propia de Windows.

También podremos encontrar otros **volúmenes** adicionales, como la unidad de DVD, memorías USB, etc., que aparecerán con nombres diferentes según el sistema operativo. En Windows aparecen como nuevas unidades raíz, por ejemplo D:\ o E:\. En OS X, aparecen como nuevas carpetas bajo la carpeta /Volumes. En Linux, aparecen como nuevas carpetas bajo la carpeta /mnt ("mount", montar).

LA BARRA DIAGONAL INVERSA (\) Y LA BARRA DIAGONAL (/).

En Windows, las rutas se escriben usando barras disgonales inversas (\) como separadores entre los nombres de las carpetas. Pero en otros sistemas operativos, se usa la barra diagonal (/). Si queremos que nuestros programas funcionen para todos los sistemas operativos, tendremos que hacer que nuestro código sea capaz de manejar ambos casos.

Afortunadamente, esto es fácil de resolver con la función `os.path.join()`. Si le pasamos las cadenas del archivo individual y de los nombres de las carpetas de la ruta a ese archivo, esta función nos devolverá una cadena con una ruta que use los separadores correctos. Escribe lo siguiente en el shell interactivo:

```
>>> import os
>>> os.path.join('usr', 'bin', 'spam', 'file.ext')
'usr\\bin\\spam\\file.ext'
```

Como aquí estamos ejecutando este ejemplo en un ordenador con Windows, la instrucción `os.path.join('usr','bin','spam','file.ext')` ha devuelto `'usr\\bin\\spam\\file.ext'`. (Notar que las barras inversas aparecen dos veces porque están en su forma de carácter de escape). Si hubiésemos ejecutado este ejemplo en un ordenador con OS X o Linux, la cadena que habríamos obtenido sería `'usr/bin/spam/file.ext'`.

La función `os.path.join` es útil cuando necesitamos crear cadenas para los nombres de archivos. Estas cadenas se les pasarán como argumentos a varias de las funciones relacionadas con archivos que veremos en este capítulo. Por ejemplo, el siguiente código une los nombres de una lista de nombres de archivo al final de un nombre de carpeta:

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
        print(os.path.join('C:\\Usuarios\\Alejandro', filename))


C:\Usuarios\Alejandro\accounts.txt
C:\Usuarios\Alejandro\details.csv
C:\Usuarios\Alejandro\invite.docx
```

EL DIRECTORIO DE TRABAJO ACTUAL.

Todo programa que se ejecute en nuestro ordenador tiene un **directorio de trabajo actual**, o **cwd** (current working directory). Cualquier nombre de archivo o ruta que no comience con la carpeta raíz se asume que está bajo el directorio de trabajo actual. Podemos obtener una cadena con el directorio de trabajo actual mediante la función `os.getcwd()`, y cambiarlo con la función `os.chdir()`. Escribe lo siguiente en el shell interactivo:

```
>>> import os
>>> os.getcwd()
'C:\\Users\\Usuario\\AppData\\Local\\Programs\\Python\\Python37-32'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

En mi ordenador, el directorio de trabajo actual es C:\Users\Usuario\AppData\Local\Programs\Python\Python37-32, por lo que el nombre de archivo project.docx se refiere a C:\Users\Usuario\AppData\Local\Programs\Python\Python37-32\project.docx. Si cambiamos el directorio actual a C:\Windows, el nombre project.docx se intepretará como C:\Windows\project.docx.

Python mostrará un mensaje de error si intentamos cambiar el cwd a un directorio que no existe:

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] El sistema no puede encontrar el archivo
especificado: 'C:\\ThisFolderDoesNotExist'
```

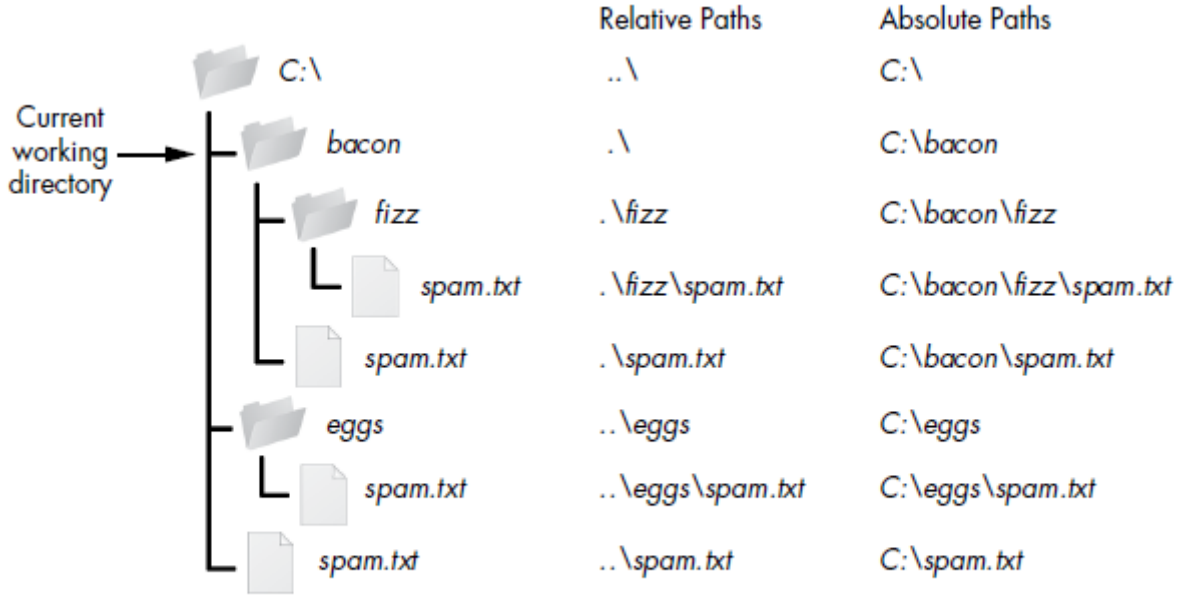
RUTAS ABSOLUTAS VS. RUTAS RELATIVAS.

Hay dos formas de especificar una ruta a un archivo:

- Una **ruta absoluta**, la cual siempre empieza con la carpeta raíz.
- Una **ruta relativa**, que se escribe relativa al directorio de trabajo actual del programa.

También existen las carpetas punto (.) y punto-punto (..). No son carpetas reales, sino nombres especiales que pueden usarse al escribir una ruta. Un solo punto para un nombre de carpeta es una abreviatura para "esta carpeta". Dos puntos (punto-punto) significa "la carpeta superior".

La figura muestra un ejemplo de una estructura de carpetas y archivos en un ordenador. Cuando el directorio de trabajo actual se fija a C:\bacon, las rutas relativas al resto de carpetas y archivos son los mostrados en la figura:



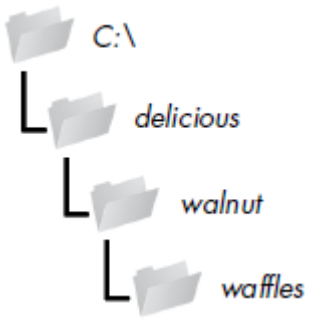
The relative paths for folders and files in the working directory C:\bacon

El .\ al principio de las rutas relativas es opcional. Por ejemplo, .\spam.txt y spam.txt se refieren al mismo archivo.

CREAR NUEVAS CARPETAS CON OS.MAKEDIRS().

Nuestros programas pueden crear nuevas carpetas con la función `os.makedirs()`. Escribe lo siguiente en el shell interactivo:

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```



Esta instrucción creará en nuestro ordenador no solo una carpeta C:\delicious, sino también una carpeta walnut dentro de C:\delicious, y una carpeta waffles dentro de C:\delicious\walnut. Esto es, `os.makedirs()` creará cualquier carpeta que sea necesaria para asegurar la existencia de la ruta competa especificada. La figura muestra esta jerarquía de carpetas.

8.2. EL MÓDULO OS.PATH.

El módulo `os.path` contiene muchas funciones útiles relacionadas con nombres de archivos y rutas. Por ejemplo, ya hemos usado la función `os.path.join()` para construuir rutas que funcionen en cualquier sistema operativo. Como `os.path` es un módulo dentro del módulo `os`, podemos importarlo escribiendo simplemente `import os`. Siempre que nuestros programas necesiten trabajar con archivos, carpetas, o rutas a archivos, podremos consultar los ejemplos de esta sección. (La documentación completa del módulo `os.path` está disponible en la web de Python, en http://docs.python.org/3/library/os.path.html).

> NOTA: La mayoría de los ejemplos de esta sección necesitarán el módulo `os`, así que debemos recordar importarlo al principio de todos los programas que escribamos, y cada vez que arranquemos el IDLE. En caso contrario, obtendremos el mesaje de error `NameError: name 'os' is not defined`.

MANEJAR RUTAS ABSOLUTAS Y RELATIVAS.

El módulo `os.path` proporciona funciones para devolver la ruta absoluta de una ruta relativa, y para comprobar si una cierta ruta es una ruta absoluta:

- La función `os.path.abspath(`*`path`*`)` devolverá una cadena con la ruta absoluta de la ruta pasada como argumento. Esta es una forma fácil de convertir una ruta relativa en una absoluta.
- La función `os.path.isabs(`*`path`*`)` devolverá `True` si el argumento es una ruta absoluta, y `False` si es una ruta relativa.
- La función `os.path.relpath(`*`path`*`, `*`start`*`)` devolverá una cadena con una ruta relativa desde la ruta *`start`* hasta la ruta *`path`*. Si no se proporciona *`start`*, se usa el directorio de trabajo actual como ruta de inicio.

Escribe lo siguiente en el shell interactivo:

```
>>> os.path.abspath('.')
'C:\\Users\\Alejandro\\AppData\\Local\\Programs\\Python\\Python37-32'
>>> os.path.abspath('.\\Scripts')
'C:\\Users\\Alejandro\\AppData\\Local\\Programs\\Python\\Python37-32\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

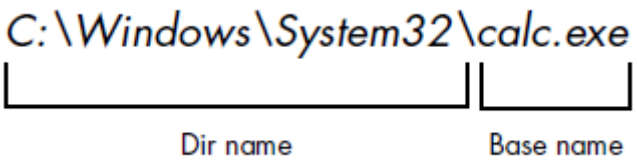
Como C:\Users\Alejandro\AppData\Local\Programs\Python\Python37-32 era el directorio de trabajo actual cuando se llamó a `os.path.abspath()`, la carpeta "punto" representa la ruta absoluta `'C:\Users\Alejandro\AppData\Local\Programs\Python\Python37-32'`.[5]

Escribe las siguientes llamadas a `os.path.relpath()` en el shell interactivo:

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
'C:\\Users\\Alejandro\\AppData\\Local\\Programs\\Python\\Python37-32'
```

[5] Como tu sistema probablemente tenga carpetas y archivos muy diferentes al mío, no podrás seguir textualmente los ejemplos de esta sección. Trata de seguirlos según las carpetas que existan en tu ordenador.

La función `os.path.dirname(path)` devolverá un cadena con todo lo que venga antes de la última barra en el argumento path. La función `os.path.basename(path)` devolverá una cadena con todo lo que venga detrás de la última barra en el argumento *path*. A modo de ejemplo, la figura muestra el **nombre de directorio** (dir name) y el **nombre base** (base name) de la ruta a la calculadora de Windows:



Por ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

Si necesitamos tener juntos (pero independientes) el nombre de directorio y el nombre base de una ruta, podemos llamar a la función `os.path.split()` para obtener una tupla con estas dos cadenas, como por ejemplo:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

También podríamos crear la misma tupla llamando a las funciones `os.path.dirname()` y `os.path.basename()` y ubicando sus valores de retorno en una tupla:

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
('C:\\Windows\\System32', 'calc.exe')
```

, pero `os.path.split()` es un buen atajo si necesitamos ambos valores.

La función `os.path.split()` no devuelve una lista de cadenas para cada carpeta de la ruta que le pasemos como argumento. Si eso es lo que queremos, debemos usar el método de cadena `split()` y dividir la cadena allá donde encuentre un separador de carpetas (`\` ó `/`). El separador de carpetas correcto para el sistema operativo bajo el que se está ejecutando el programa se almacena en la variable `os.path.sep`. A modo de ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> calcFilePath.split(os.path.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

OBTENER LOS TAMAÑOS DE LOS ARCHIVOS Y LOS CONTENIDOS DE LAS CARPETAS.

Una vez sabemos cómo manejar las rutas de los archivos, podemos empezar a reunir información acerca de los archivos y las carpetas. El módulo `os.path` proporciona funciones para hallar el tamaño de un determinado archivo en bytes, y para conocer los archivos y carpetas que están contenidos dentro de una carpeta dada.

- La función `os.path.getsize(path)` devolverá el tamaño en bytes del archivo en la ruta pasada en el argumento *path*.

- La función `os.listdir(path)` devolverá una lista de cadenas con los nombres de los archivos y de las carpetas presentes en la ruta pasada en el argumento *path*. (Notar que esta función está contenida en el módulo `os`, y no en el módulo `os.path`).

Vamos a probar estas dos funciones en el shell inetractivo:

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
26112
>>> os.listdir('C:\\Users')
['Alejandro', 'All Users', 'Default', 'Default User', 'Default.migrated',
'desktop.ini', 'Public', 'TEMP', 'UpdatusUser']
```

Como podemos ver, el programa calc.exe ocupa 26112 bytes en mi ordenador, y en la carpeta C:\Users hay 9 archivos y carpetas. Si queremos hallar el tamaño total de todos los archivos en una determinada carpeta, podemos usar juntas las funciones `os.path.getsize()` y `os.listdir()`:

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
      totalSize = totalSize +
      os.path.getsize(os.path.join('C:\\Windows\\System32', filename))

>>> print(totalSize)
1394216397
```

Conforme iteramos sobre cada nombre de archivo en la carpeta C:\Windows\System32 (carpeta que está muy cargada de archivos y otras carpetas), el tamaño total de la variable `totalSize` se incrementa en el tamaño de cada archivo. Notar que cuando llamamos a `os.path.getsize()`, usamos `os.path.join()` para unir el nombre de la carpeta con el nombre del archivo actual. El entero que devuelve `os.path.getsize()` se suma al valor de `totalSize`. Después de iterar a través de todos los archivos, imprimimos `totalSize` para ver el tamaño total de la carpeta C:\Windows\System32.

COMPROBAR LA VALIDEZ DE UNA RUTA.

Muchas funciones de Python lanzarán un error si les proporcionamos una ruta que no existe. El módulo `os.path` proporciona funciones para comprobar si existe una cierta ruta, y para saber si se trata de un archivo o una carpeta.

- La función `os.path.exists(path)` devolverá `True` si el archivo o carpeta referido en la ruta pasada como argumento existe, y devolverá `False` si no existe.
- La función `os.path.isfile(path)` devolverá `True` si la ruta pasada como argumento existe y es un archivo, y devolverá `False` en cualquier otro caso.
- La función `os.path.isdir(path)` devolverá `True` si la ruta pasada como argumento existe y es una carpeta, y devolverá `False` en cualquier otro caso.

Esto es lo que obtenemos cuando probamos estas funciones en el shell interactivo:

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
```

```
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

También podemos saber si actualmente hay una unidad de DVD o un lápiz de memoria conectados al ordenador comprobándolo con la función `os.path.exists()`. Por ejemplo, si quisiéramos comprobar si hay una memoria USB en el volumen E:\ de mi ordenador Windows, podríamos hacer lo siguiente:

```
>>> os.path.exists('E:\\')
False
```

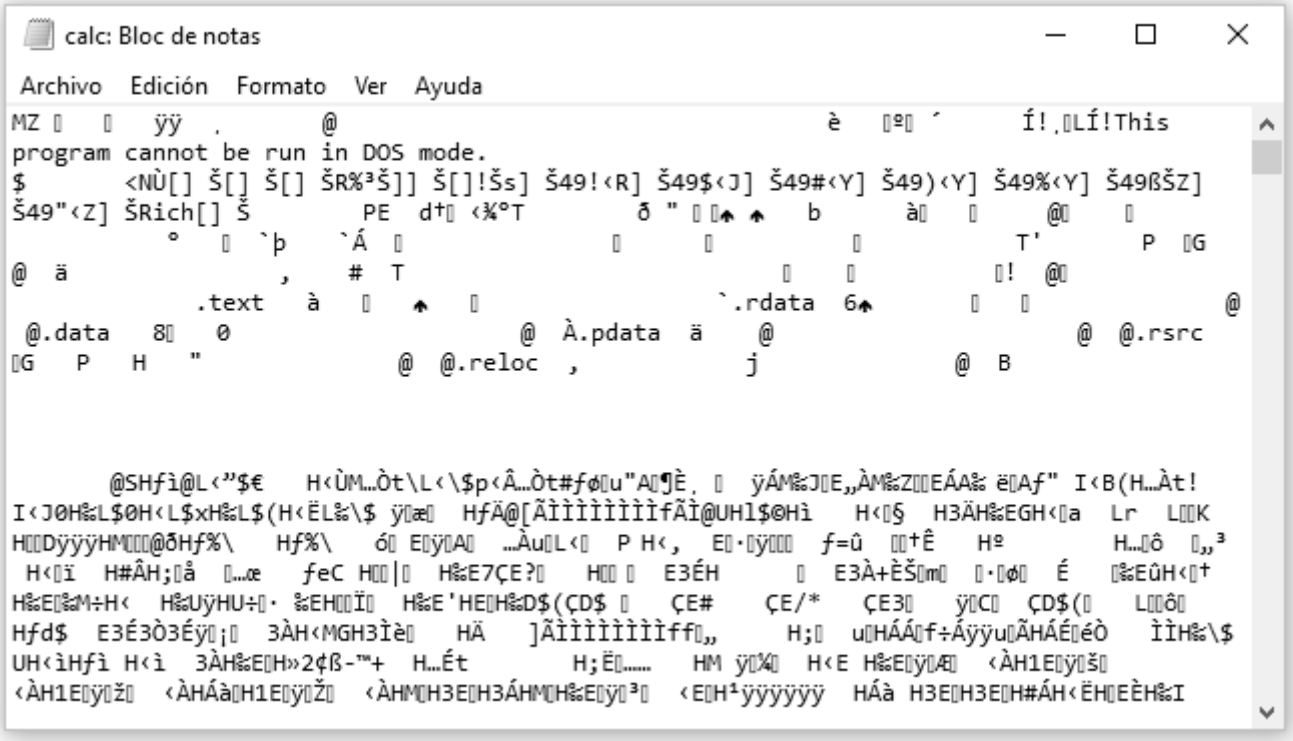
La función devuelve `False` porque ahora mismo no tengo un lápiz de memoria insertado en mi ordenador

8.3. LECTURA Y ESCRITURA DE ARCHIVOS.

Ahora que ya sabemos cómo trabajar con carpetas y rutas relativas, somos capaces de especificar la localización de los archivos de los que vamos a leer o en los que vamos a escribir.

Las funciones de las que hablaremos en esta sección solo aplican a archivos de "texto plano" (plaintext). Los **archivos de texto plano** son aquellos que sólo contienen caracteres de texto básicos y no incluyen formato, esto es, información acerca del tipo de fuente, tamaño, color, etc. Algunos ejemplos de archivos de texto plano son los archivos con extensión *.txt* o los archivos de texto de Python con extensión *.py*. Este tipo de archivos puede abrirse con el Bloc de Notas (Notepad) de Windows, o con la aplicación TextEdit de OS X. Nuestros programas de Python también pueden leer fácilmente el contenido de un archivo de texto plano y tratarlos como un valor tipo cadena ordinario.

El resto de archivos son los denominados **archivos binarios**, e incluyen los archivos de procesadores de texto como Microsoft Word, los archivos PDF, las imágenes, las hojas de cálculo, y los programas ejecutables. Si abrimos un archivo binario con el Bloc de Notas o con el TextEdit, parecerá un galimatías de caracteres y símbolos sin sentido, ya que no son simples archivos de texto plano. A modo de ejemplo, la figura muestra el archivo calc.exe de Windows abierto con el Bloc de Notas:



Como cada tipo de archivo binario debe manejarse de forma distinta, aquí no veremos cómo leer o escribir archivos binarios directamente. (Sin embargo, existen módulos que nos permitirán trabajar con archivos

binarios de forma mucho más sencilla, como el módulo `shelve`. Hablaremos más de él en capítulos posteriores).

Hay tres pasos que siempre debemos seguir a la hora de leer o escribir archivos en Python:

1) Llamar a la función `open()` para obtener un objeto tipo `File`.
2) Llamar a los métodos `read()` o `write()` sobre el objeto `File`.
3) Cerrar el archivo llamando al método `close()` sobre el objeto `File`.

ABRIR ARCHIVOS CON LA FUNCIÓN OPEN().

Para abrir un archivo con la función `open()`, debemos pasarle una cadena que indique la ruta (absoluta o relativa) en la que está localizado el archivo en nuestro ordenador. La función `open()` retorna un objeto de tipo `File`.

A modo de ejemplo, crea un archivo de texto con el Bloc de Notas llamado *hello.txt*. Escribe `Hello World!` como contenido de este archivo y guárdalo en tu carpeta de trabajo. A continuación (si estás usando Windows) escribe lo siguiente en el shell interactivo:

```
>>> helloFile = open('C:\\Users\\Alejandro\\Dropbox\\PYTHON\\PROGRAMAS\\hello.txt')
```

(En tu caso, deberás poner la ruta en la que tengas el archivo en *tu* ordenador). Este comando abrirá el archivo en modo "lectura de texto plano", o **modo lectura**, para abreviar. Cuando abre un archivo en modo lectura, Python solo nos permite leer datos desde el archivo; no nos permite escribir o modificarlo de ninguna forma. El modo lectura es el modo por defecto para todos los archivos que abramos en Python. Pero si no queremos fiarnos de los valores por defecto de Python, podemos especificar de forma explícita el modo en el que abrimos un archivo, pasando el valor tipo cadena `'r'` como segundo argumento de la función `open()`. Así, `open('C:\\Users\\Alejandro\\Dropbox\\PYTHON\\PROGRAMAS\\hello.txt')` y `open('C:\\Users\\Alejandro\\Dropbox\\PYTHON\\PROGRAMAS\\hello.txt', 'r')` hacen exactamente la misma cosa (esto es, abrir el archivo en modo lectura).

Como hemos indicado antes, la llamada a `open()` devuelve un objeto de tipo `File`. Un objeto tipo `File` representa un archivo en nuestro ordenador; se trata simplemente de otro tipo de datos en Python, como lo son las listas y los diccionarios. En el ejemplo previo guardamos el objeto `File` en la variable `helloFile`. Ahora, cuando queramos leer de este archivo o escribir sobre este archivo, lo haremos llamando a los métodos del objeto `File` guardado en la variable `helloFile`.

LEER LOS CONTENIDOS DE LOS ARCHIVOS.

Ahora que disponemos de un objeto `File`, podemos comenzar a leerlo. Si queremos leer el contenido completo de un archivo como un valor de tipo cadena, debemos usar el método `read()` del objeto `File`. Vamos a leer el contenido del archivo *hello.txt* que almacenamos en la variable `helloFile`. Escribe lo siguiente en el shell interactivo:

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

Si nos imaginamos el contenido de un archivo como un único valor de tipo cadena, el método `read()` devuelve la cadena que está almacenada en el archivo.

Alternativamente, podemos usar el método `readline()` para leer línea a línea un archivo de texto. Por ejemplo, crea un archivo llamado *sonnet29.txt* en la misma carpeta que el archivo *hello.txt*, y escribe el siguiente texto dentro de él:

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Asegúrate de separar las cuatro líneas con saltos de línea. A continuación, escribe lo siguiente en el shell interactivo:

```
>>> sonnetFile = open('C:\\Users\\Usuario\\Dropbox\\PYTHON\\PROGRAMAS\\
sonnet29.txt')
>>> sonnetFile.readline()
"When, in disgrace with fortune and men's eyes,\n"
>>> sonnetFile.readline()
'I all alone beweep my outcast state,\n'
>>> sonnetFile.readline()
'And trouble deaf heaven with my bootless cries,\n'
>>> sonnetFile.readline()
'And look upon myself and curse my fate,'
>>> sonnetFile.readline()
''
```

Observa que cada nueva llamada al método `readline()` sobre el objeto `sonnetFile` devuelve la siguiente línea del archivo de texto. Cuando ya no quedan más líneas que leer, `readline()` devuelve una cadena vacía. A modo de ejercicio, escribe un programa que use un bucle para leer línea a línea todas las líneas del archivo *sonnet29.txt*.

Como última opción, podemos usar el método `readlines()` para obtener una *lista* de valores de tipo cadena, donde cada línea del texto será una cadena de la lista. Escribe este código en el shell interactivo:

```
>>> sonnetFile = open('C:\\Users\\Alejandro\\Dropbox\\PYTHON\\PROGRAMAS\\
sonnet29.txt')
>>> sonnetFile.readlines()
["When, in disgrace with fortune and men's eyes,\n", 'I all alone beweep
my outcast state,\n', 'And trouble deaf heaven with my bootless cries,\n'
, 'And look upon myself and curse my fate,']
>>>
```

Notar que cada uno de los valores tipo cadena termina con un carácter de nueva línea (`\n`), excepto la última línea del archivo. ¿Para qué sirve el método `readlines()`? Bueno, normalmente es más fácil trabajar con una lista de cadenas que con una única gran cadena.

ESCRIBIR EN ARCHIVOS.

Python nos permite escribir nuevo contenido en un archivo de texto de la misma forma que la función `print()` "escribe" cadenas por pantalla. Sin embargo, no podremos escribir sobre un archivo que hayamos abierto en modo lectura. En vez de ello, debemos abrir el archivo en el modo "escritura de texto plano" o en el modo "agregación de texto plano" (también llamados **modo escritura** (write mode) y **modo agregación** (append mode), para abreviar).

El modo escritura sobreescribirá el archivo existente y comenzará desde cero, igual que cuando reescribimos el valor de una variable con un nuevo valor. Para abrir un archivo en modo escritura, debemos pasar `'w'` como segundo argumento a la función `open()`. Por otro lado, el modo agregación añadirá el texto

al final del archivo existente. Podemos imaginarnos este proceso como el acto de añadir una variable a una lista. Para abrir un archivo en modo agregación, debemos pasar `'a'` como segundo argumento a la función `open()`.

Si el nombre que le pasamos a la función `open()` no existe, tanto el modo escritura como el modo agregación crearán un nuevo archivo en blanco. Después de leer o escribir un archivo, hemos de llamar al método `close()` antes de poder abrir el archivo otra vez.

Vamos a ver cómo se usan todas estas instrucciones. Escribe el siguiente código en el shell interactivo:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

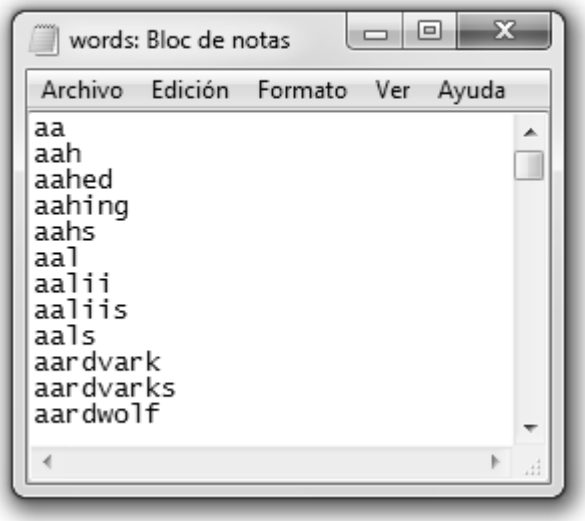
En primer lugar, abrimos el archivo *bacon.txt* en modo escritura. Como el archivo *bacon.txt* no existe, Python crea un archivo en blanco. La llamada al método `write()` sobre el archivo abierto pasándole como argumento `'Hello world! /n'` nos permite escribir esta cadena en el archivo, y obtener el número de caracteres escritos, incluyendo el carácter de nueva línea (`/n`). Después, cerramos el archivo.

Si no queremos reemplazar el contenido que ya hemos escrito, sino añadir texto nuevo a los contenidos ya existentes, abrimos el archivo en modo agregación. Escribimos `'Bacon is not a vegetable.'` en el archivo y lo cerramos. Finalmente, para imprimir los contenidos del archivo por pantalla, abrimos el archivo en su modo de lectura por defecto, llamamos al método `read()`, almacenamos el objeto `File` resultante en `content`, cerramos el archivo, e imprimimos `content`.

Observar que el método `write()` no añade automáticamente un carácter de nueva línea al final de la cadena, cosa que sí hace la función `print()`. En el caso del método `write()`, debemos añadir ese carácter a mano nosotros mismos.

LECTURA DE LISTAS DE PALABRAS.

Para algunos de los ejercicios de este capítulo necesitaremos una lista de palabras en inglés. Hay muchas listas de palabras disponibles en la web; una de ellas es la lista de palabras recopilada y distribuida para el dominio público por Grady Ward como parte del proyecto Moby (ver https://en.wikipedia.org/wiki/Moby_Project). Se trata de una lista de 113809 palabras que se consideran válidas en crucigramas y otros juegos de palabras. Este archivo lo tienes disponible, con el nombre *words.txt*, en el archivo comprimido con el material del curso, y también on-line en la web http://greenteapress.com/thinkpython2/code/words.txt.



Como este archivo es texto plano podemos abrirlo con un editor de textos cualquiera, pero también con Python:

```
>>> fin = open('C:\\Users\\Usuario\\Dropbox\\PYTHON\\PROGRAMAS\\words.txt')
```

Una vez abierto el archivo (en su modo de lectora por defecto), podemos leerlo palabra a palabra con el método `readline()`:

```
>>> fin.readline()
'aa\n'
>>> fin.readline()
'aah\n'
```

Notar que el método `readline()` devuelve una cadena con la palabra y el carácter de salto de línea, porque en el archivo las palabras están en líneas distintas. Si este carácter nos molesta, podemos omitirlo usando el método `strip()` propio de los valores tipo cadena.

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

También podemos usar un objeto `File` como parte de un bucle. El siguiente programa lee el archivo *words.txt* e imprime cada palabra, una por línea.

```
fin = open('C:\\Users\\Usuario\\Dropbox\\PYTHON\\PROGRAMAS\\words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

A modo de ejercicio, escribe un programa que lea el archivo *words.txt* y que imprima solamente aquellas palabras con más de 20 caracteres.

8.4. GUARDAR VARIABLES CON EL MÓDULO SHELVE.

En nuestros programas podemos guardar el contenido de una variable en un archivo binario (binary shelf file) usando el módulo `shelve`. De esta forma, nuestros programas podrán recuperar los datos desde el disco duro para volver a cargarlos en variables. El módulo `shelve` nos permitirá añadir las funcionalidades de "Guardar" (Save) y "Abrir" (Open) a nuestros programas. Por ejemplo, si ejecutásemos un programa e introdujésemos algunos ajustes de configuración, podríamos guardar esos ajustes en un archivo, para después hacer que el programa los cargue la próxima vez que se ejecute.

Escribe lo siguiente en el shell interactivo:

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

Para leer y escribir datos usando el módulo `shelve`, primero debemos importarlo. A continuación llamamos a `shelve.open()` pasándole un nombre de archivo, y almacenamos el objeto `shelf` devuelto en una variable. Entonces, podemos hacer cambios al objeto `shelf` como si de un diccionario se tratase. Cuando hayamos acabado, llamamos al método `close()` sobre el objeto `shelf`. En este ejemplo, guardamos el objeto `shelf` en la variable `shelfFile`. Creamos una lista llamada `cats` y escribimos

`shelfFile['cats'] = cats` para almacenar la lista en `shelfFile`, en la forma de un valor asociado a la clave `'cats'` (igual que en un diccionario). Para terminar, llamamos a `close()` sobre `shelfFile`.

Al ejecutar este código en Windows, veremos que aparecen tres nuevos archivos en el directorio de trabajo actual: *mydata.bak*, *mydata.dat*, y *mydata.dir*. En OS X solo se creará un archivo *mydata.db*.

Estos archivos binarios contienen los datos que hemos guardado en nuestro en nuestro objeto `shelf` (literalmente, en nuestra "estantería"). El formato de estos archivos binarios no es importante; lo único que necesitamos saber es qué es lo que hace el módulo `shelve` (almacenar datos en un archivo), pero no cómo lo hace.

Nuestros programas pueden usar el módulo shelve para abrir con psterioridad y recuperar datos de estos archivos shelf. Los valores `shelf` no tienen que abrirse en modo lectura o escritura, pueden hacer ambas cosas desués de ser abiertos. Escribe lo siguiente en el shell interactivo:

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

Aquí, abrimos el archivo shelf para comprobar si nuestros datos se han guardado correctamente. Al escribir `shelfFile['cats']` obtenemos la misma lista que guardamos antes, y así sabemos que la lista se ha guardado adecuadamente. Entonces, llamamos a `close()` para cerrar el archivo.

Al igual que los disccionarios, los valores `shelf` tienen unos métodos `keys()` y `values()` que devuelven valores tipo lista de las claves y los valores del objeto `shelf`. Como estos métodos devuelven valores tipo lista, y no listas auténticas, deberíamos pasárselos a la función `list()` para conventirlos en listas. Escribe lo siguiente en el shell interactivo:

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

Los archivos de texto plano es útil para crear archivos que leeremos después en un editor de textos, como el Bloc de Notas o TextEdit. Pero si queremos guardar datos de nuestros programas Python, debemos usar el módulo `shelve` para almacenarlos en archivos binarios.

8.5. GUARDAR VARIABLES CON LA FUNCIÓN PPRINT.PFORMAT().

Recordar de la sección 6.2 que la función `pprint.pprint()` permite una impresión con formato por pantalla de los contenidos de una lista o de un diccionario, miestras que la función `pprint.pformat()` devuelve ese mismo texto formateado como una cadena (en vez de imprimirlo por pantalla). Pues bien, esta cadena formateada no solo es sfácil de leer, sino también constituye un código Python sintácticamente correcto. Imaginar que tenemos un diccionario almacenado en una variable, y que queremos guardar esta variable y sus contenidos para usos futuros. La función `pprint.pformat()` nos dará una cadena que podemos escribir a un archivo .py. Este archivo será un módulo propio que podemos importar siempre que queremos usar la variable que hemos almacenado dentro de él.

A modo de ejemplo, escribe lo siguente en el shell interactivo:

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc':
'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

En este ejemplo, importamos `pprint` para poder usar la función `pprint.pformat()`. Tenemos una lista de diccionarios guardada en la variable `cats`. Para que la lista de la variable `cats` siga disponible incluso después de cerrar el shell, usamos `pprint.pformat()` para devolverla como una cadena. Una vez disponemos de los datos en `cats` en forma de cadena, es fácil escribir la cadena en un archivo, al que llamaremos *myCats.py*.

Al final, los módulos que importamos mediante una sentencia `import` también son programas Python. Cuando la cadena devuelta por `pprint.pformat()` se guarda en un archivo .py, el archivo es un módulo que puede importarse igual que cualquier otro.

Y como los programas Python son simplemente archivos de texto con la extensión .py, nuestros programas Python pueden incluso generar otros programas Python. Y podemos importar estos archivos en nuestros programas:

```
>>> import myCats
>>> myCats.cats
[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]
>>> myCats.cats[0]
{'desc': 'chubby', 'name': 'Zophie'}
>>> myCats.cats[0]['name']
'Zophie'
```

La ventaja de crear un archivo .py (en vez de guardar variables con el módulo `shelve`) es que, como se trata de un archivo de texto, los contenidos de este archivo los puede leer y modificar cualquiera con un simple editor de texto. Sin embargo, y para la mayoría de aplicaciones, la mejor estrategia para guardar variables a un archivo es usar el módulo `shelve`. Esto es así porque sólo los tipos de datos básicos, como los enteros, flotantes, cadenas, listas, y diccionario, pueden ser escritos como texto plano en un archivo. Los objetos `File`, por ejemplo, no pueden codificarse como texto.

8.6. PROYECTO: JUEGO DE PREGUNTAS Y RESPUESTAS.

Este juego plantea al usuario una serie de preguntas con varios opciones de respuesta sobre un determinado tema. El tema que aquí usaremos es la mafia, y el nombre del tema es "un tema que no podrás rechazar" (An Episode You Can't Refuse). Tidas las preguntas están relacionadas directa o indirectamente con la mafia.

Lo bueno de este juego es que las preguntas de un cierto tema están almacenadas en un archivo independiente del archivo del juego. De esta forma, cualuiqera con un edito de texto plano puede crear sus propias preguntas sobre el tema que elijan (zoología, fórmula 1, cine, etc.). La única precaución a tomar es que el archivo de preguntas que el programa leerá, *trivia.txt*, debe estar en la misma carpeta que el archivo del programa.

ESTRUCTURA DEL ARCHIVO DE PREGUNTAS.

Antes de empezar a programar, es importante decidir la esructura que tendrá el archivo de preguntas *trivia.txt*. La primera línea del archivo es el tema de las preguntas. El resto del archivo consiste en bloques de ocho líneas, un bloque por pregunta. Podemos tener tantos bloques (y por consiguiente, preguntas) como queramos.

He aquí una representación genérica de un bloque:

```
<category>
<question>
<answer 1>
<answer 2>
<answer 3>
<answer 4>
<correct answer>
<explanation>
```

Y aquí tenemos el principio del archivo que hemos creado para el juego:

```
An Episode You Can't Refuse
On the Run With a Mammal
Let's say you turn state's evidence and need to "get on the lamb." If you wait
/too long, what will happen?
You'll end up on the sheep
You'll end up on the cow
You'll end up on the goat
You'll end up on the emu
1
A lamb is just a young sheep.
The Godfather Will Get Down With You Now
Let's say you have an audience with the Godfather of Soul. How would it be
/smart to address him?
Mr. Richard
Mr. Domino
Mr. Brown
Mr. Checker
3
James Brown is the Godfather of Soul.
```

Recordar que la primera línea es el título del tema de las preguntas. Las siguientes siete líneas son para la primera pregunta. Y las siguientes siete líneas son para la segunda pregunta. Así pues, `On the Run With a Mammal` es la categoría de la primera pregunta. La categoría es solo una introducción a la pregunta que viene después. La siguiente línea, `Let's say you turn state's evidence and need to "get on the lamb." If you wait /too long, what will happen?`, es la primera pregunta del juego. Las siguientes cuatro líneas, `You'll end up on the sheep`, `You'll end up on the cow`, `You'll end up on the goat`, `You'll end up on the emu`, son las cuatro rspuestas posibles entre las que elegirá el usuario. La siguiente línea, `1`, es el número de la respuesta correcta. En este caso, la respuesta correcta es `You'll end up on the sheep`. A siguiente línea, `A lamb is just a young sheep.`, explica la razón por la que la respuesta correcta es correcta.

Notar que hemos añadido una barra (`/`) en un par de líneas. Eso lo hacemos para representar un cambio de línea, ya que Python nmo cambia de líena automáticamente cuando imprime texto por pantalla. Cuando el programa lea una línea del archivo de texto, reemplezará todas la barras en esa línea por el carácter de nueva línea (ver función `next_line()` más adelante).

LA FUNCIÓN OPEN_FILE().

Lo primero que vamos a hacer es definir una función `open_file()`, que recibirá un nombre de archivo y un modo (ambos en forma de cadena) y devolverá el objeto `File` correspondiente. Vamos a definir una función porque usaremos manejo de excepciones para controlar los errores que ocurren, por ejemplo, cuando el nombre de archivo pasado como argumento no existe. El error lanza Python cuando no se ha podido abrir un archivo es un error de tipo `IOError`, y este es el error que deberemos manejar.
Si ocurre una excepción es porque ha habido un problema al abrir el archivo de preguntas, y no tiene sentido continuar con el programa, así que imprimiremos el correspondiente mensaje, y llamaremos a la función `sys.exit()`. Esta función lanza una excepción que fuerza la finalización del programa. Solo deberíamos usar `sys.exit()` como último recurso, cuando debamos finalizar un programa. Recordar que para usar esta función, debemos importar el módulo `sys`.

Escribe lo siguiente en el editor de archivos, y guarda el programa como `trivia_challenge.py`:

```
# Trivia Challenge
# Trivia game that reads a plain text file

import sys

def open_file(file_name, mode):
    """Open a file."""
    try:
        the_file = open(file_name, mode)
    except IOError:
        print("Unable to open the file", file_name, "Ending program.\n")
        input("\n\nPress the enter key to exit.")
        sys.exit()
    else:
        return the_file
```

LA FUNCIÓN NEXT_LINE().

A continuación definimos la función `next_line()`, para leer línea a línea un archivo de texto. Esta función recibe un objeto tipo `File` y devuelve la siguiente línea de texto en el archivo:

```
def next_line(the_file):
    """Return next line from the trivia file, formatted."""
    line = the_file.readline()
    line = line.replace("/", "\n")
    return line
```

Esta función también efectúa un poco de formateado a la línea antes de devolverla. El método de cadena `replace()` reemplaza todas la barras (`/`) por caracteres de salto de línea (`\n`). Esto debemos hacerlo porque al imprimir texto por pantalla, Python no cambia de lñinea automáticamente. Nuestra función le otorga al creador del archivo de preguntas un cierto control en el formato del texto. Esa persona puede indicar dónde aparecerán los cambios de línea escribiendo una barra `/` allá donde los desee. De esta forma podrán evitar que las palabras aparezcan partidas en dos líneas.

LA FUNCIÓN NEXT_BLOCK().

La función `next_block()` lee el bloque de ocho líneas para la siguiente pregunta. Esta función recibe un objeto `File` y devuelve cuatro cadenas, y una lista de cadenas. Las cuatro cadenas son la categoría, la pregunta, la respuesta correcta, y la explicación, mientras que la lista contiene las cuatro cadenas de las cuatro posibles respuestas a la pregunta:

```
def next_block(the_file):
    """Return the next block of data from the trivia file."""
    category = next_line(the_file)

    question = next_line(the_file)

    answers = []
    for i in range(4):
        answers.append(next_line(the_file))

    correct = next_line(the_file)
    if correct:
        correct = correct[0]

    explanation = next_line(the_file)

    return category, question, answers, correct, explanation
```

Al leer la respuesta correcta y almacenarla en la cadena `correct`, comprobamos que la cadena no esté vacía (`if correct:`), y en tal caso, nos quedamos solamente con el primer carácter de la cadena (`correct = correct[0]`). De esta forma nos aseguramos de quedarnos solamente con el número de la respuesta correcta, y no con posibles espacios en blanco adicionales que el creador del archivo de preguntas haya podiso insertar tras él.

Si la función llega al final del archivo de preguntas, la lectura de una nueva línea devolverá una cadena vacía. Por lo tanto, cuando el programa llegue al final del archivo *trivia.txt*, `category` recibirá una cadena vacía. Esta comprobación la hacemos en la función `main()` del programa (la función que implementa el bucle principal del juego), y en ese caso, el juego termina.

LA FUNCIÓN WELCOME()

La función `welcome()` da la bienvenida al jugador, y muestra el título del tema de las preguntas. La función recibe el título del tema en forma de cadena, y lo imprime junto con un mensaje de bienvenida:

```
def welcome(title):
    """Welcome the player and get his/her name."""
    print("\t\tWelcome to Trivia Challenge!\n")
    print("\t\t", title, "\n")
```

CONFIGURAR EL JUEGO.

Ahora creamos la función `main()`, que se encarga de implementar el bucle principal. E la primera parte de la función, abrimos el archivo de preguntas, obtenemos el título del tema de las preguntas (la primera línea del archivo), le damos la bievenida al jugador, y fijamos su puntuación a `0`.

```
def main():
    trivia_file = open_file("trivia.txt", "r")
    title = next_line(trivia_file)
    welcome(title)
    score = 0
```

FORMULAR UNA PREGUNTA.

Ahora, y todavía en la función `main()`, leemos el primer bloque de líneas para la primera pregunta, y las almacenamos en variables. A continuación, comenzamos un bucle while, que continuará formulando preguntas mientras `category` no sea una cadena vacía. Si `category` es una cadena vacía, es que hemos

llegado al final del archivo de preguntas, y no volveremos a entrar al bucle. Para hacer una pregunta, primero imprimimos la categoría de la pregunta, luego la pregunta es sí misma, y finalmente las cuetro posibles respuestas. Añade lo siguiente a la función `main()`:

```
# get first block
category, question, answers, correct, explanation = next_block(trivia_file)
while category:
    # ask a question
    print(category)
    print(question)
    for i in range(4):
        print("\t", i + 1, "-", answers[i])
```

OBTENER LA RESPUESTA DEL JUGADOR.

Seguimos dentro de la función `main()`. A continuación, obtenemos la respuesta del jugador:

```
# get answer
answer = input("What's your answer?: ")
```

COMPROBAR LA RESPUESTA.

Una vez tenemos la respuesta del jugador, la comparamos con la respuesta correcta. Si coinciden, felicitamos al jugador y su puntuación aumenta en un punto. Si no coinciden, indicamos al jugador que ha errado. En ambos casos, mostramos la explicación, que describe por qué razón la respuesta correcta es efectivamente correcta. Por último, mostramos la puntuación actual del usuario:

```
# check answer
if answer == correct:
    print("\nRight!", end=" ")
    score += 1
else:
    print("\nWrong.", end=" ")
print(explanation)
print("Score:", score, "\n\n")
```

OBTENER LA SIGUIENTE PREGUNTA.

Después, llamamos a la función `next_block()` para obtener el bloque de cadenas para la próxima pregunta. Si no hay más preguntas, `category` será una cadena vacía y el bucle no continúa.

```
# get next block
category, question, answers, correct, explanation = next_block(trivia_file)
```

FINALIZAR EL JUEGO.

Al salir del bucle, cerramos el archivo de preguntas y mostramos la puntuación del jugador:

```
trivia_file.close()

print("That was the last question!")
print("You're final score is", score)
```

ARRANCAR LA FUNCIÓN MAIN().

Ya fuera de la función `main()`, la última línea de código simplemente llama a la función `main()`.

8.7. PROYECTO: GENERAR EXÁMENES ALEATORIOS.

Imagina que eeres un profesor de geografía con 35 estudiantes en clase, y que quieres examinarles sobre capitales de países. Además, hay un grupo de estudiantes que suelen copiar, por lo que no puedes ponerles el mismo examen a todos ellos. Nos gustaría aleatorizar el orden de las preguntas de forma que cada examen sea distinto de los demás, para hacer imposible que los estudiantes puedan copiarse echando un vistazo al examen del compañero. Evidentemente, hacer los 35 exámenes a mano sería una tarea larga y aburrida. Por suerte, nuestros conocimientos de Python nos ayudarán a automatizar esta tarea.

Nuestro programa debería hacer lo siguiente:

- Crear 35 exámenes diferentes.
- Crear 50 preguntas tipo test para cada examen, en orden aleatorio.
- Para cada pregunta, el programa debe proporcionar la respuesta correcta y otras tres respuestas incorrectas, en orden aleatorio.
- Escribir los exámenes en 35 archivos de texto.
- Escribir la plantilla de respuestas para cada uno de los exámentes, en otros tantos archivos de texto.

Esto significa que nuestro código debe hacer lo siguiente:

- Almacenar los países y sus respectivas capitales en un diccionario.
- Llamar a las funciones y procedimientos `open()`, `write()`, y `close()` para los archivos de los exámenes y de las plantillas de respuestas.
- Usar la función `random.shuffle()` para aleatorizar el orden de las preguntas y de las respuestas de opción múltiple.

PASO 1: ALMACENAR LOS DATOS EN UN DICCIONARIO.

El primer paso es crear el esquieleto del programa, y rellenarlo con los datos del examen. Crea un nuevo archivo llamado `randomQuizGenerator.py`, y escribe algo parecido a lo siguiente:

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

import random

# The quiz data. Keys are countries and values are their capitals.
capitals = {'Spain': 'Madrid', 'France': 'Paris', 'United Kingdom': 'London',
'Germany': 'Berlin', 'Belgium': 'Brussels', 'Denmark': 'Copenhagen',
'Italy': 'Rome', 'Netherlands': 'Amsterdam', 'Iceland': 'Reykjavik',
'Norway': 'Oslo', 'Sweden': 'Stockholm', 'Finland': 'Helsinki', 'Austria':
'Vienna', 'Switzerland': 'Bern', 'Hungary': 'Budapest', 'Czech Replublic':
'Prague', 'Poland': 'Warsaw', 'Russia': 'Moscow', 'Bulgaria':
'Sofia', 'Romania': 'Bucharest', 'Ukraine': 'Kiev', 'Belarus':
'Minsk', 'Slovakia': 'Bratislava', 'Turkey': 'Ankara', 'Brasil':
'Brasilia', 'Venezuela': 'Caracas', 'Colombia': 'Bogota', 'Peru':
'Lima', 'Argentina': 'Buenos Aires', 'Chile': 'Santiago', 'Canada': 'Ottawa',
'Nicaragua': 'Managua', 'Honduras': 'Tegucigalpa',
'United States': 'Washington DC', 'Cuba': 'La Habana', 'Costa Rica': 'San
Jose',
'China': 'Beijing', 'India': 'New Delhi', 'Vietman': 'Hanoi',
'Thailand': 'Bangkok', 'Malaysia': 'Kuala Lumpur', 'Iran':
'Tehran', 'Irak': 'Bagdad', 'Egypt': 'El Cairo', 'Sirya':
'Damascus', 'Morocco': 'Rabat', 'Kenya': 'Nairobi',
'South Africa': 'Pretoria', 'Australia': 'Camberra', 'New Zealand':
'Wellington'}
```

```
# Generate 35 quiz files.
for quizNum in range(35):
    # TODO: Create the quiz and answer key files.
    # TODO: Write out the header for the quiz.
    # TODO: Shuffle the order of the states.
    # TODO: Loop through all 50 states, making a question for each.
```

(1) Como este programa ordenará aleatoriamente las preguntas y las respuestas, necesitamos importar el módulo `random` para hacer uso de sus funciones. (2) La variable `capitals` contiene un diccionario con algunos países como claves y sus capitales como valores. (3) Y como queremos crear 35 exámenes, el código genera los archivos para los exámenes y para las plantillas de respuestas (lo que hemos marcado momnetáneamente con comentarios `TODO` (por hacer)). Por supuesto, este número puede cambiarse para generar cualquier número de archivos que necesitemos.

PASO 2: CREAR EL ARCHIVO DEL EXAMEN Y DESORDENAR LAS PREGUNTAS.

Vamos a empezar a rellenar algunos bloques `TODO`.

El código dentro del bucle se repetirá `35` veces (una vez por cada examen), por lo que solo debemos ocuparnos de un examen cada vez dentro del bucle. Primero creamos el archivo para el examen actual. Este archivo necesita un nombre de archivo único, y también debería tener algún tipo de encabezado estándar dentro de él, que le indique al alumno que escriba su nombre, la fecha, y el curso y grupo. A continuación debemos obtener una lista de países en orden aleatorio, ue usarmos después para crear las preguntas y las respuestas para el examen.

Añade lo siguiente al programa `randomQuizGenerator.py`:

```
# Generate 35 quiz files.
for quizNum in range(35):
    # Create the quiz and answer key files.
  ❶ quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')
  ❷ answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w')

    # Write out the header for the quiz.
  ❸ quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
    quizFile.write((' ' * 20) + 'Capitals Quiz (Form %s)' % (quizNum + 1))
    quizFile.write('\n\n')

    # Shuffle the order of the countries.
    countries = list(capitals.keys())
  ❹ random.shuffle(countries)

    # TODO: Loop through all 50 countries, making a question for each.
```

Los nombres de archivo para los exámenes serán `capitalsquiz<N>.txt`, donde `<N>` es un número único que proviene de `quiznum`, el contador del bucle `for`. El archivo para la plantilla de respuestas del examen `capitalsquiz<N>.txt` se guardará en un archivo llamado `capitalsquiz_answer<N>.txt`. En cada pasada del bucle, el **marcador** `%s` en `'capitalsquiz%s.txt'` y `'capitalsquiz_answers%s.txt'` será reemplazado por `(quizNum + 1)`, por lo que los primeros archivos se crearán con los nombres `capitalsquiz1.txt` y `capitalsquiz_answers1.txt`. Estos archivos se crearán con sendas llamadas a funciones `open()` en (1) y (2), con `'w'` como segundo argumento para abrirlos en modo escritura.

Las sentencias `write()` en (3) crean un encabezado en el examen para que lo rellenen los alumnos. Finalmente, en (4) se crea una lista aleatoria de países con ayuda de la función `random.shuffle()`, que reordena aleatoriamente los valores de cualquier lista que se le pase como argumento.

PASO 3: CREAR LAS DISTINTAS RESPUESTAS PARA CADA PREGUNTA.

Ahora debemos generar las distintas respuestas para cada una de las preguntas, a saber, 4 posibles respuestas de la A a la D, donde solo una de ellas es la correcta. Aquí necesitaremos otro bucle for, encargado de generar el contenido de cada una de las 50 preguntas del examen. A continuación, un tercer bucle for anidado generará las respuestas posibles para cada pregunta. Completa el código del programa como sigue:

```
# Loop through all 50 countries, making a question for each.
for questionNum in range(50):
    # Get right and wrong answers.
  ❶ correctAnswer = capitals[countries[questionNum]]
  ❷ wrongAnswers = list(capitals.values())
  ❸ del wrongAnswers[wrongAnswers.index(correctAnswer)]
  ❹ wrongAnswers = random.sample(wrongAnswers, 3)
  ❺ answerOptions = wrongAnswers + [correctAnswer]
  ❻ random.shuffle(answerOptions)

    # TODO: Write the question and answer options to the quiz file.
    # TODO: Write the answer key to a file.
```

(1) La respuesta correcta es fácil de obtener: está almacenada como un valor en el diccionario `capitals`. Este bucle itarará a través de los países en la lista aleatorizada `countries`, desde `countries[0]` hasta `countries[49]`, encontrará cada país en `capitals`, y guardará la capital correspondiente de ese país en `correctAnswer`.

La lista de todas las respuestas incorrectas que son posibles es algo más difícil de obtener. Podemos conseguirla duplicando *todos* los valores del diccionario `capitals` (2), borrando la respuesta correcta (3), y seleccionando tres valores aleatorios de esta lista (4). La función `random.sample()` nos facilita enormemente hacer esta selección. Su primer argumento es la lista de la cual queremos elegir algunos valores aleatorios, y el segundo argumento es el número de valores aleatorios que queremos obtener. La lista completa de posibles respuestas a una pregunta dada es una combinación de estas tres respuestas incorrectas y la respuesta correcta (5). Finalmente, en (6), se reordenan estas cuatro posibles respuestas, para que la respuesta correcta no esté siempre en la posición D.

PASO 4: ESCRIBIR LOS CONTENIDOS DE LOS ARCHIVOS DEL EXAMEN Y DE LA PLANTILLA DE RESPUESTAS.

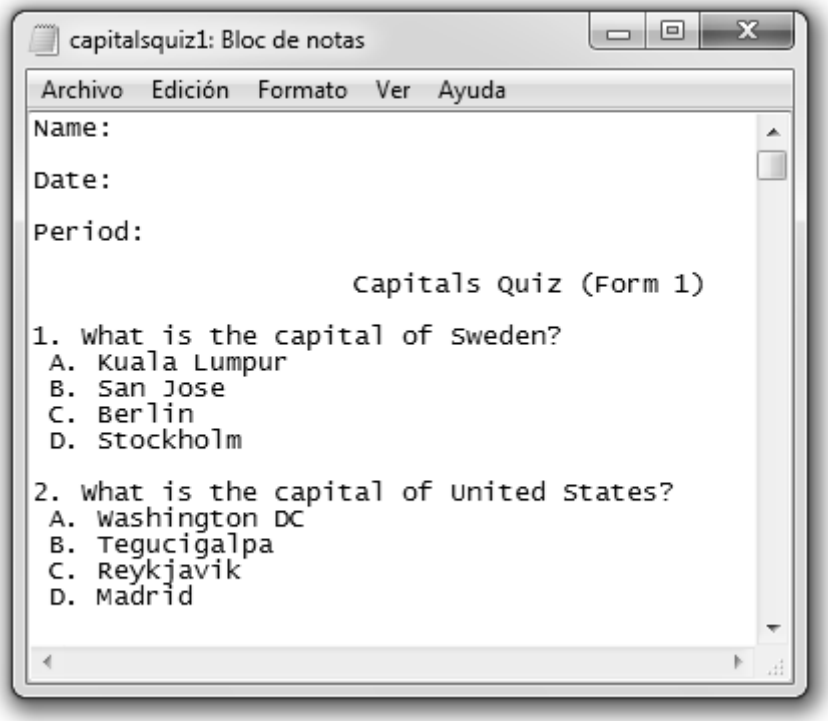
Todo lo que queda por hacer es escribir las preguntas en el archivo del examen y las respuestas en el archivo de la plantilla. Completa el programa con lo siguiente:

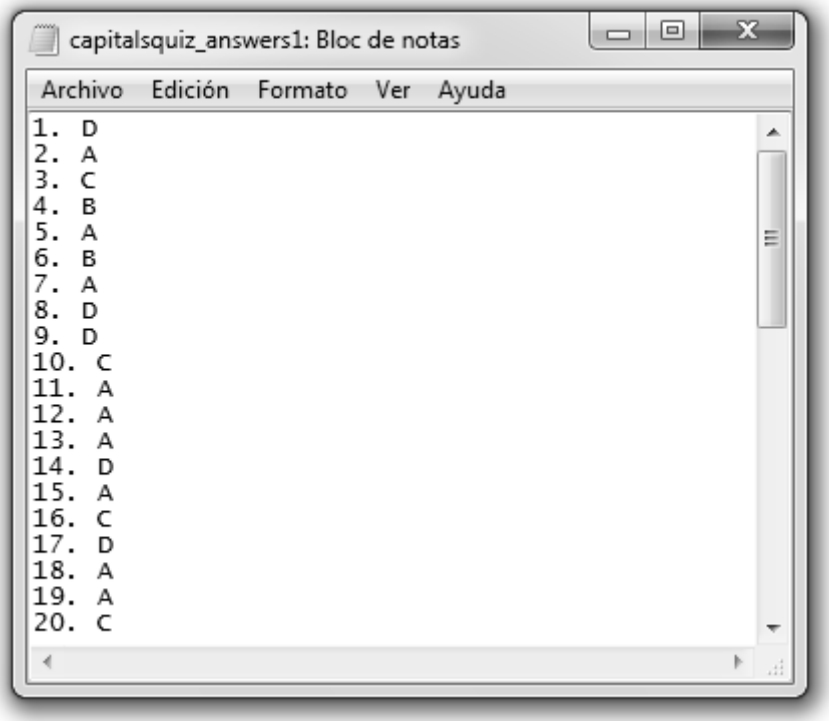
```
# Loop through all 50 countries, making a question for each.
for questionNum in range(50):
    # Get right and wrong answers.
    correctAnswer = capitals[countries[questionNum]]
    wrongAnswers = list(capitals.values())
    del wrongAnswers[wrongAnswers.index(correctAnswer)]
    wrongAnswers = random.sample(wrongAnswers, 3)
    answerOptions = wrongAnswers + [correctAnswer]
    random.shuffle(answerOptions)
```

```
    # Write the question and the answer options to the quiz file.
    quizFile.write('%s. What is the capital of %s?\n'
                   % (questionNum + 1, countries[questionNum]))
❶   for i in range(4):
❷       quizFile.write(' %s. %s\n' % ('ABCD'[i], answerOptions[i]))
    quizFile.write('\n')

    # Write the answer key to a file.
❸   answerKeyFile.write('%s. %s\n' %(questionNum + 1,
                                     'ABCD'[answerOptions.index(correctAnswer)]))

quizFile.close()
answerKeyFile.close()
```

En (1) tenemos un bucle `for` que itera desde `0` hasta `3` para escribir las distintas opciones de respuesta contenidas en la lista `answerOptions`. La expresión `'ABCD'[i]` en (2) trata la cadena `'ABCD'` como un vector que se evaluará a `'A'`, `'B'`, `'C'`, y `'D'` a cada pasada del bucle. En la última línea, en (3), la expresión `answerOptions.index(correctAnswer)` hallará el índice entero de la respuesta correcta en la lista desordenada `answerOptions`, y `'ABCD'[answerOptions.index(correctAnswer)]` se evaluará a la letra de la respuesta correcta. Esta letra será la que escribiremos en el archivo de plantilla.

Vamos a probar el programa. Cambia el primer bucle `for` de `35` a `1` repeticiones, para que solo genere un archivo de preguntas y respuestas. Tras ejecutar el programa, en nuestra carpeta de trabajo aparecerán los archivos *capitalsquiz1.txt* y *capitalsquiz1_answers.txt*, que serán similares a los mostrados en las figuras (aunque obviamente, las preguntas y respuestas serán diferentes a las que mostramos aquí, debido a la aleatorización del orden de las preguntas y de las respuestas posibles):





8.8. PROYECTO: PORTAPAPELES MÚLTIPLE

Imaginar que tenemos que completar la (aburridísima) tarea de rellenar un monton de formularios (en una página web o en un software) con múltiples campos de texto. El portapapeles nos permite ahorrarnos escribir la misma cosa una y otra vez. Pero el inconveniente es que el poritapapeles sólo puede guardar una cosa al mismo tiempo. Ello implica que, si tenemos varios textos diferentes a copiar y pegar, deberíamos selecciónalos uno a uno y cipiarlos uno a uno, lo cual sería ciertamente engorroso.

Pero podemos solventar este problema escribiendo un programa Python que sea capaz de almacenar varios textos en una especie de "portapapeles múltiple". Este programa se llamaará *mcb.pyw*. "mcb" es el acrónimo de "multiclipboard". La extensión *.pyw* significa que Python no mostrará una ventana de terminal al ejecutar este programa.

Este programa almacenará cada trozo de texto del portapapeles bajo una palabra clave. Por ejemplo, al ejecutar `py mcb.pyw save spam`, el contenido actual del portapapeles se guardará con la palabra clave *spam*. Más adelente, podremos volver a cargar este texto en el portapapeles ejecutando `py mcb.pyw spam`. Y si el usuario se olvida de la palabra clave que asoció a un texto en particular, puede ejecutar `py mcb.pyw list` para copiar una lista de todas las palabras clave definidas hasta el momento.

Así pues, el programa debería hacer lo siguiente:
- Comprueba el argumento de línea de comandos.
- Si el argumento es `save`, los contenidos del portapaleles se guardan bajo la palabra clave especificada a continuación.
- Si el argumento es `list`, se copian todas las palabras clave en el portapapeles.
- En otro caso, se copia el texto asociado a la palabra clave.

Esto significa que el código deberá hacer todo esto:
- Leer el argumento de línea de comandos desde `sys.argv`.
- Leer yb escribir en el portapapeles.
- Guardar/cargar datos en/desde un archivo shelf.

Si nuestro ordenador usa Windows, podemos ejecutar fácilmente ste programa desde la ventana "ejecutar" (WIN + R) creando un archivo *mcb.bat* con el siguiente contenido:

```
@pyw.exe C:\ruta\al\archivo\mcb.pyw %*
```

PASO 1: COMENTARIOS Y CONFIGURACIÓN DEL ARCHIVO SHELF.

Comenzamos escribiendo el esqueleto del programa, con algunos comentarios y la configuración básica. Escibe lo siguiente en el editor de archivos, y guarda el programa como `mcb.pyw`:

```
#! python3
# mcb.pyw - Saves and loads pieces of text to the clipboard.
❶ # Usage: py.exe mcb.pyw save <keyword> - Saves clipboard to keyword.
# py.exe mcb.pyw <keyword> - Loads keyword to clipboard.
# py.exe mcb.pyw list - Loads all keywords to clipboard.

❷ import shelve, pyperclip, sys

❸ mcbShelf = shelve.open('mcb')

# TODO: Save clipboard content.

# TODO: List keywords and load content.

mcbShelf.close()
```

(1) Es una práctica habitual en Python añadir información de uso del programa mediante comnetarios al principio del código. De esta forma, si alguna vez nos olvidamos de cómo se ejecuta el programa, basta con leer estos comentarios a modo de recordatorio. (2) A continuación, importamos los módulos. Las acciones de copiar y pegar requiere el módulo `pyperclip`, y leer argumentos de línea de comandos requiere el módulo `sys`. El módulo `shelve` también será necesario: Siempre que el usuario quiera guardar un nuevo texto del portapapeles, lo guardaremos en un archivo shelf. Después, cuando el usuario quiera volver a pegar ese texto en el portapapeles, abrimereos el archivo shelf y lo cargaremos en el programa. El nombre del archivo shelf vendrá precedido del prefijo *mcb* (3)

PASO 2: GUARDAR EL CONTENIDO DEL PORTAPAPELES CON UNA PALABRA CLAVE.

El programa hace cosas distintas dependiendo de si el usuario quiere guardar texto del portapapeles bajo una palabra clave, cargar texto al portapapeles, o listar las palabras clave ya existentes. Vamos a trabajar en el primer caso. Modifica el programa para que se parezca a lo siguiente:

```
import shelve, pyperclip, sys

mcbShelf = shelve.open('mcb')

# Save clipboard content.
❶ if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
❷     mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
❸     # TODO: List keywords and load content.

mcbShelf.close()
```

(1) Si el primer argumento de línea de comando (que siempre estará en el índice 1 de la lista `sys.argv`) es `'save'`, el segundo argumento de línea de comando es la palabra clave para el contenido actual del portapapeles. (2) Esa palabra clave se usará como clave de `mcbShelf`, y el valor asociado será el texto que hay actualmente en el portapapeles.

Si solo hay un argumento de línea de comandos, podemos asumir que será `'list'` o una palabra clave para cargar contenido en el portapapeles. Implementaremos este código más adelante, y añadimos un recordatorio de ello con un comentario `TODO`.

PASO 3: LISTAR LAS PALABRAS CLAVE Y CARGAR EL CONTENIDO DE UNA PALABRA CLAVE.

Finalmente, vamos a implementar los dos casos restantes: El usuario quiere cargar al portapepeles contenido de una palabra clave, o quiere una lista de todas las palabras clave disponibles. Modifica el programa para incluir lo siguiente:

```
# Save clipboard content.
if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
    mcbShelf[sys.argv[2]] = pyperclip.paste()
elif len(sys.argv) == 2:
    # List keywords and load content.
 ❶ if sys.argv[1].lower() == 'list':
 ❷     pyperclip.copy(str(list(mcbShelf.keys())))
    elif sys.argv[1] in mcbShelf:
 ❸     pyperclip.copy(mcbShelf[sys.argv[1]])

mcbShelf.close()
```

(1) Si solo hay un argumento de línea de comandos, primero comprobamos si se trata de `'list'`. (2) Si es así, se copia en el portapapeles una representación en forma de cadena de la lista de claves del archivo shelf. El usuario podrá pegar esa lista en un editor de texto para leerla.

(3) En caso contrario, podemos asumir que el argumento de línea de comnados es una palabra clave. Si esa palabra clave existe como clave en el archivo shelf `mcbShelf`, cargamos el valor asociado en el portapepeles.

Y ya hemos terminado. Para lanzar este programa bajo Windows, debemos consultar el anexo B de estos apuntes.

Recordar el programa gestor de contraseñas de la sección 7.3 que almacenaba las contraseñas en un diccionario. Actualizar las contraseñas implicaba cambiar el código fuente del programa. Esto no es lo más adecuado, porque los usuarios "normales" (esto es, sin conocimientos de Python) no quieren cambar el código fuente para actualizar su software. Además, cada vez que modificamos el código fuente de un programa, coremos el riesgo de introducir malfuncionamientos en el programa. Pero si almacenamos los datos del programa en un archivo distinto al del código, podemos hacer nuestros programas más fáciles de usar y más resistentes frente a malfuncionamientos.

8.9. EJERCICIOS DEL CAPITULO 8.

Ejercicio 8.1. Crea un archivo llamado *newFile.txt*. En primer lugar, añade una nueva línea al archivo, por ejemplo, `This is the first line of newFile.txt text file`. A continuación, imprime por pantalla el contenido del archivo. Guarda el programa como `Ejer8.1.py`.

Ejercicio 8.2. Escribe una función que lea las n primeras líneas de un archivo de texto. Guarda el programa como `Ejer8.2.py`.

Ejercicio 8.3. Escribe una función que agregue una cadena de texto al final de un archivo de texto, y que lo imprima por pantalla. Guarda el programa como `Ejer8.3.py`.

Ejercicio 8.4. Escribe un programa que lea línea a línea un archivo de texto, y que guarde cada línea como una cadena en una lista. Guarda el programa como `Ejer8.4.py`.

Ejercicio 8.5. Escribe un programa que lea línea a línea un archivo de texto, y que guarde todo el contenido en una única variable de tipo cadena. Guarda el programa como `Ejer8.5.py`.

Ejercicio 8.6. Escribe un programa que cuente ell número de líneas de un archivo de texto. Guarda el programa como `Ejer8.6.py`.

Ejercicio 8.7. Escribe un programa que halle la líena más larga de un archivo de texto, y la imprima por pantalla. Guarda el programa como `Ejer8.7.py`.

Ejercicio 8.8. Escribe un rograma que cuente el número de palabras de un archivo de texto. Guarda el programa como `Ejer8.8.py`.

Ejercicio 8.9. Escribe un programa que copie los contenids de un archivo origen a otro archivo destino. Guarda el programa como `Ejer8.9.py`.

Ejercicio 8.10. Escribe un programa que combine cada línea de un primer archivo con la línea correspondiente en un archivo de destino. Escribe el resultado en un tercer archivo, inicialmente vacío. Por simplicidad, los dos archivos originles deben tener exactamente el mismo número de líneas escritas. Guarda el programa como `Ejer8.10.py`.

Ejercicio 8.11. Escribe un programa que lea una líena laeatorio de un archivo, y la imprima por pantalla. Guarda el programa como `Ejer8.11.py`.

Ejercicio 8.12. Filtros.
Un tipo especial de programa es aquel que lee un texto línea a línea de un archivo de entrada, y que realiza un pequeño precesado conforme escribe esas líneas en un archivo de salida. A modo de ejemplo, podría numerar esas líneas, insetar líneas en blanco después de haber pasado 60 líneas (para permitir una impresión crrecta en papel), extraer algunas columnas específicas de cada línea en el archivo fuente, o imprimir únicamente aquellas líenas que cntengan una cadena específica. A este tipo de programas se les llama **filtros**.
a) Escribe un filtro que copie un archivo origen en un archivo destino, omitiendo cualquier línea que comience con una almohadilla (#). Guarda el programa como `Ejer8.12a.py`.
b) Escribe un filtro que copie un archivo origen en un archivo destino, añadiendo delante de cada línea el número de línea, comenzando por el 1. Guarda el programa como `Ejer8.12b.py`.
c) Escribe un programa que implemente un filtro a tu elección. Al principio del programa, escribe con comentarios qué filtro aplica. Guarda el programa como `Ejer8.12c.py`.

Ejercicio 8.13. Escribe un programa que deshaga la numeración realizada por el ejercicio 8.12(b), esto es, que lea un archivo donde las líenas estén numeradas, y produzca otro archivo en el que los números de línea hayan desaparecido. Guarda el programa como `Ejer8.13.py`.

Ejercicio 8.14. Escribe un programa que lea un archivo de texto y escriba un nuevo archivo en el que las líneas del archivo original están en orden inverso (por ejemplo, la primera línea del archivo original se

convierte en la última línea del archivo nuevo, la segunda líena del archivo original se convierte en la penúltima del archivo nuevo, etc.). Guarda el programa como `Ejer8.14.py`.

Ejercicio 8.15. Constantes físicas.
El archivo *constants.txt* contiene algunas constantes físicas fundamentales, junto con su valor y unidades. Queremos cargar esta tabla en un diccionario llamado `constants`, donde las claves sean los nombres de las constantes. Por ejemplo, `constants['gravitational constant']` almacena el valor de la constante de gravitación de Newton, $6{,}67259 \times 10^{-11}$. Escribe una función que lea e interprete el archivo de texto, y devuelva el diccionario. Guarda el programa como `Ejer8.15.py`

Ejercicio 8.16. Diccionario anidado (2).
El archivo *human_evolution.txt* guarda información sobre varias especies humanas acerca de su época, altura, peso, y capacidad cerebral. Haz un programa que lea este archivo y almacene los datos en un diccionario anidado llamado `humans`. La clave de `humans` será el nombre de la especie (por ejemplo, `homo erectus`), y los valores son diccionarios con claves para `height`, `weight`, `brain volumen`, y `when` (esta última para la época en la que vivió esa especie). Por ejemplo, `humans['homo neanderthalensis'][ 'mass']` debería ser igual a `'55 - 70'`. Además, haz que el programa escriba por pantalla una tabla con los datos almacenados en ese diccionario. Guarda el programa como `Ejer8.16.py`

Ejercicio 8.17. Los archivos primenumbers.txt y happynumbers.txt contienen todos los números primos y números felices hasta el 1000.[6] Escribe un programa que encuentre qué números de ambos archivos son tanto primos como felices, y guárdalos en una lista. Guarda el programa como `Ejer8.17.py`

Ejercicio 8.18. El siguiente archivo de texto se llama *studentData.txt*, y contiene una línea por cada alumno de una clase. El nombre del alumno es el primer elemento de cada línea, y está seguido por las notas (sobre 100) de sus exámenes. El número de notas puede ser diferente para cada alumno:

```
joe 10 15 20 30 40
bill 23 16 19 22
sue 8 22 17 14 32 17 24 21 2 9 11 17
grace 12 28 21 45 26 10
john 14 32 25 16 89
```

a) Usando el archivo de texto *studentsData.txt*, escribe un programa que imprima por pantalla los nombres de los estudiantes que tienen más de seis notas. Guarda el programa como `Ejer8.18a.py`.
b) Usando el archivo de texto *studentsData.txt*, escribe un programa que calcule la nota media para cada estudiante, y que imprima por pantalla los nombres de los estudiantes junto con sus notas medias. Guarda el programa como `Ejer8.18b.py`.
c) Usando el archivo de texto *studentsData.txt*, escribe un programa que calcule las notas máxima y mínima para cada estudiante, y que imprima por pantalla los nombres de los estudiantes junto con esas dos notas. Guarda el programa como `Ejer8.18c.py`.

Ejercicio 8.19. Escribe una función que reciba una letra como argumento. La función comienza leyendo un archivo de texto, y construyendo una lista donde cada elemento sea una palabra del archivo. La función debe devolver una nueva lista con todas las palabras que contengan la letra pasada como argumento, y mostrar por pantalla cuántas palabras hay en la lista que contengan esa letra. Guarda el programa como `Ejer8.19.py`.

[6] Los números felices son un concepto real en matemáticas. Si sientes curiosidad, busca en Internet qué crterios debe cumplir un número para ser "feliz".

Ejercicio 8.20. Correos electrónicos.
Imaginar que queremos enviar por correo electrónico la misma invitación a muchos destinatarios. El cuerpo del correo siempre es el mismo. Sólo cambia el nombre (y la dirección) del destinatario. En vez de escribir un correo electrónico a cada destinatario, vamos a desarrollar un programa que use un archivo con la plantilla común para el cuerpo del correo (*body.txt*) y un archivo con una lista de nombres (*names.txt*), que fusionaremos para formar todos los correos a enviar.

El archivo *names.txt* contiene todos los nombres de los destinatarios en líneas separadas. Por su parte, el archivo *body.txt* contiene el mensaje de la invitación. El objetivo es crear un correo por cada destinatario (que incluya un saludo personalizado con el nombre del destinatario, más el texto de la invitación), y guardar cada correo en un archivo independiente.

Asegúrate de crear los archivos *body.txt* y *names.txt* (este último, con 3 o 4 nombres).

PISTA: Abre ambos archivos en modo lectura, e itera a lo largo de ellos usando sendos bucles `for`. A cada pasada del bucle, crea un nuevo archivo *.txt* para el correo (cuyo nombre de archivo sea el nombre del destinatario), y fusiona adecuadanmente los contenidos de *name.txt* y *body.txt* para generar el contenido de cada correo. Usa el método `strip()` para limpiar saltos de línea, espacios en blanco, etc. Finalmente, escribe el contenido del correo en los nuevos archivos usando el método `write()`.

Guarda el programa como `Ejer8.20.py`.

Ejercicio 8.21. Modifica el programa del portapapeles múltiple para que tenga un argumento de línea de comandos `delete <keyword>` que permita borrar una palabra clave (y su contenido) del shelf. Añade también un argumento de línea de comnados `delete` que permita borrar *todas* las palabras clave. Guarda el programa como `Ejer8.21.py`.

Ejercicio 8.22. Mad Libs
Crea un program de tipo Mad Libs que lea un archivo de texto y que le permita al usuario añadir su propio texto allá donde encuentre las palabras ADJETIVO, SUSTANTIVO, ADVERBIO, o VERBO en el archivo de texto. Por ejemplo, el archivo de texto podría ser:

```
El hombre ADJETIVO caminó hacia el SUSTANTIVO y entonces VERBO. Un SUSTANTIVO
que pasaba por allí se sintió ADJETIVO ante este hecho.
```

El programa halla estas ocurrencias y le pide al usuario palabras para reemplazarlas:

```
Escribe un adjetivo:
descomunal
Escribe un sustantivo:
oso
Escribe un verbo:
ladró
Escribe sustantivo:
camión
Escribe un adjetivo:
hambriento
```

El programa debería crear un archivo de texto con este contenido:

```
El hombre descomunal caminó hacia el oso y entonces ladró. Un camión que pasaba
por allí se sintió hambriento ante este hecho.
```

Guarda el programa como `Ejer8.22.py`.

Ejercicio 8.23. Queremos determinar si una cierta palabra está presente en una lista de palabras. Por supuesto, podríamos resolver este problema mediante el operador `in`, pero la búsqueda sería muy lenta, porque la función buscaría a través de la lista de palabras una tras otra. Si las palabras de nuestra lista están en orden alfabético, podemos acelerar el proceso mediante una **búsqueda por bisección** (o **búsqueda binaria**), similar a lo que hacemos cuando buscamos una palabra en el diccionario. Comenzamos por la mitad, y comprobamos si la palabra buscada viene antes o después de la palabra que está en la mitad de la lista. Si viene antes, buscamos en la primera mitad de la lista de la misma forma; si viene después, buscamos en la segunda mitad. En cualquier caso, siempre reducimos el espacio de búsqueda a la mitad. Si la lista contiene 113809 palabras, nos llevará unas 17 iteraciones encontrar la palabra o concluir que no está allí. Escribe una función que reciba una lista de palabras ordenada y una palabra a buscar, y que devuelva `True` si la palabra está en la lista, o `False` en caso contrario. Guarda el programa como `Ejer8.23.py`.

9. PROGRAMACIÓN ORIENTADA A OBJETOS.

La **programación orientada a objetos** (POO) es una de las técnicas más efectivas para escribir programas. En POO escribimos **clases** que representan cosas del mundo real, y creamos **objetos** basados en esas clases. Al escribir una clase, definimos el comportamiento general de una cierta categoría de objetos. Después, al crear objetos individuales pertenecientes a esa clase, cada nuevo objeto creado está equipado automáticamente con ese comportamiento general. A partir de entonces, cada objeto de esa clase puede evolucionar de una forma particular y específica. Aunque suene un poco abstracto, resulta sorprendente descubrir la cantidad de situaciones que pueden modelarse mediante objetos.

Al acto de crear un objeto a partir de una clase se le llama *instanciación*, y nosotros trabajaremos con **instancias** de una clase. En este capítulo aprenderemos a escribir clases y a crear objetos (o instancias) de esas clases. Especificaremos el tipo de información que puede almacenarse en las instancias, y definiremos las acciones que pueden realizar esas instancias. También escribiremos clases que extiendan la funcionalidad de ciertas clases existentes, de forma que las clases que sean similares puedan compartir el código eficientemente. Finalmente, aprenderemos a guardar clases en módulos para poder importar esas clases en nuestros programas.

9.1. PRINCIPIOS BÁSICOS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS.

La **programación orientada a objetos** (POO) es muy útil para representar cosas del mundo real en nuestros programas. Como las cosas del mundo real, los **objetos** poseen ciertas características (llamadas **atributos**) y pueden realizar ciertas acciones (llamadas **métodos**). Por ejemplo, si queremos modelar un coche mediante un objeto, sus atributos podrían ser su marca, su modelo, su matrícula, o su velocidad instantánea, y sus métodos podrían incluir la habilidad de acelerar, frenar, o encender sus luces.

Los objetos se crean (o se **instancian**) a partir de una definición común llamada **clase**, esto es, el código que define sus atributos y sus métodos. Las clases son como las plantillas de los objetos (o **instancias**) que creamos a partir de ellas.[7] Una clase *no* es un objeto, sino una plantilla o diseño para instanciar un nuevo objeto. Una vez escrita la clase, podemos crear muchas instancias a partir de ella. Como resultado, todas las instancias creadas a partir de esa clase tendrán una misma estructura básica.

Sin embargo, aunque dos instancias de la misma clase compartan una estructura básica común, también pueden poseer distintos valores de sus atributos. Así, por ejemplo, podríamos tener dos coches creados a partir de la misma clase, pero con dos valores distintos de su maca, modelo, matrícula, y velocidad instantánea.

Puede que por el momento todos estos conceptos nos parezcan un galimatías sin sentido. En esta sección solo pretendemos dar una perspectiva general de la POO. De ahora en adelante veremos cómo se materializan todos estos conceptos en Python, y cuál es su utilidad.

9.2. CREAR Y USAR UNA CLASE.

En POO podemos modelar casi cualquier cosa usando clases. Todas las clases suelen incluir **métodos**, esto es, las acciones que las distintas instancias de esa clase pueden realizar. Vamos a comenzar escribiendo una clase muy sencilla, llamada `Dog`, que represente un perro (pero no un perro en particular, sino un perro cualquiera). Esta clase le dirá a Python cómo crear un objeto que represente a un perro. Después de

[7] En POO es habitual usar los términos **objeto** e **instancia** indistintamente, ya que se refieren a la misma cosa. Lo mismo haremos en estos apuntes.

escribir esta clase, la usaremos para crear instancias (objetos) individuales, cada una de las cuales representará un perro específico.

Para empezar, definiremos un perro extremadamente simple que solo tendrá una habilidad: ponerse a ladrar. Escribe lo siguiente en una nueva ventana del editor de archivos, y guárdalo como `dog1.py`:

```
class Dog():
    #A simple attempt to model a dog.#

    def bark(self):
        print('Woof! woof! I am a new instance of class Dog.')

my_dog = Dog()
my_dog.bark()
```

DEFINIR UNA CLASE.

El programa comienza con la definición de la clase `Dog`. Esta definición comienza con la palabra reservada `class`, seguida del nombre que hemos elegido para la clase, a saber, `Dog`:

```
class Dog():
```

Notar que el nombre de la clase empieza con mayúsculas: Ésta es una convención de Python para diferenciar los nombres de las clases (en este caso, `Dog`) de los nombres de las variables que almacenan instancias de esa clase (en este caso, `my_dog`).

DEFINIR UN MÉTODO.

La última parte de la clase es la definición de un método. Como vemos, la definición de un método se parece mucho a la definición de una función:

```
def bark(self):
     print('Woof! woof! I am a new instance of class Dog.')
```

De hecho, podemos pensar que los métodos de una clase son las funciones que está asociadas a las instancias de una clase. El método `bark()` simplemente imprime la cadena `'Woof! woof! I am a new instance of class Dog.'`.

Habremos notado que `bark()` tiene un parámetro, llamado `self`, el cual parece que no estamos pasando como argumento en la llamada al método desde el programa principal. Todo **método de instancia** (o **método de objeto**) debe tener este parámetro especial, que además, siempre debe venir en primer lugar. A este parámetro siempre se le denomina `self` por convención. Los métodos utilizan el parámetro `self` como mecanismo para referirse a la propia instancia de la clase. Por el momento no nos preocuparemos más por el parámetro `self`, más adelante explicaremos su funcionamiento con mayor detalle. Si creamos un método de instancia sin ningún parámetro, Python producirá un error cuando lo llamemos. Debemos recordar que todos los métodos de instancia deben tener un primer parámetro especial llamado `self`.

CREAR UN OBJETO A PARTIR DE UNA CLASE.

Después de que hayamos escrito nuestra clase `Dog`, podemos instanciar un nuevo objeto a partir de esa clase con una sola línea de código:

```
my_dog = Dog()
```

Esta sentencia crea un nuevo objeto de la clase `Dog` y se lo asigna a la variable `my_dog`. Notar que hemos escrito dos paréntesis después del nombre de la clase en la sentencia de asignación. Estos paréntesis son obligatorios en las sentencias de creación de nuevas instancias.

El nombre de la variable donde alojamos la instancia recién creada puede ser cualquiera. Sin embargo, deberíamos evitar usar el mismo nombre de la clase (pero en minúsculas), porque ello podría producir confusiones.

LLAMAR A UN MÉTODO.

Nuestro nuevo objeto tiene un método llamado `bark()`. Este método es como cualquier otro método que ya hayamos usado en capítulos previos (como por ejemplo, los métodos propios de las listas, las cadenas, etc.). Podemos llamar a este método de la misma forma que llamamos a otro método cualquiera, usando la notación basada en puntos:

```
my_dog.bark()
```

Esta instrucción llama al método `bark()` de la instancia de la clase `Dog` recién creada, la cual está alojada en la variable `my_dog`.

CREAR UN CONSTRUCTOR.

Ya hemos visto cómo crear métodos, como por ejemplo, el método `bark()` de la clase `Dog`. Sin embargo, existe un método especial, llamado **constructor**, al que Python llama automáticamente justo cuando se crea una nueva instancia de una clase. El constructor es un método extremadamente útil, ya que sirve para establecer los valores iniciales de los **atributos** del objeto recién instanciado. Sin embargo, el constructor que escribiremos en el siguiente ejemplo no lo usaremos con esa finalidad. Abre una nueva ventana en el editor de archivos, escribe el siguiente código, y guárdalo como `dog2.py`:

```
class Dog():
    #A simple attempt to model a dog.#

    def __init__(self):
        print('A new instance of Dog has been created.')

    def bark(self):
        print('Woof! woof! I am an instance of class Dog.')

my_dog = Dog()
your_dog = Dog()

my_dog.bark()
your_dog.bark()
```

El primer bloque de código dentro de la definición de la clase es el **método constructor** (también llamado **método de inicialización**). Normalmente, podremos poner a nuestros métodos los nombres que deseemos. Sin embargo, `__init__()` es un nombre especial, y sirve para decirle a Python que ese método es el constructor de la clase. (Notar que este método tiene dos guiones bajos delante y detrás de su nombre, una convención que evita conflictos entre los métodos especiales de Python y los métodos que nosotros mismos escribimos).[8] El método constructor `__init__()` es el método al que Python llama automáticamente cada vez que se crea un nueva objeto de la clase. En este ejemplo, el método constructor

[8] Python dispone de todo un conjunto de "métodos especiales" cuyos nombres empiezan y terminan con dos guiones bajos. Un ejemplo es el método constructor `__init()__`.

simplemente informa que se ha creado una nueva instancia de la clase `Dog`, imprimiendo por pantalla la cadena `'A new instance of Dog has been created.'`.

CREAR MÚLTIPLES OBJETOS.

Una vez creada una clase, es fácil crear múltiples instancias de la misma. En el programa principal de `dog2.py` hemos creado dos instancias de la clase `Dog`:

```
my_dog = Dog()
your_dog = Dog()
```

Como resultado se crean dos instancias. Justo después de que cada instancia se haya creado, se imprime `'A new instance of Dog has been created.'` mediante su método constructor correspondiente.

Cada instancia creada es un nuevo objeto `Dog` de pleno derecho. Para comprobarlo, hemos invocado sus respectivos métodos `bark()`:

```
my_dog.bark()
your_dog.bark()
```

Aunque estas dos líneas de código impriman exactamente la misma cadena (a saber, `'Woof! woof! I am an instance of class Dog.'`), cada una es el resultado de la acción de un objeto distinto.

CREAR E INICIALIZAR ATRIBUTOS.

Los atributos son las características que poseerán todos los objetos creados a partir de una misma clase. Por ejemplo, ¿qué características tienen en común todos los perros? Bueno, suelen tener un nombre y una edad. Estas dos informaciones (`name` y `age`) constituirán los atributos de la clase `Dog`. Abre una nueva ventana en el editor de archivos, escribe el siguiente código, y guárdalo como `dog3.py`:

```
class Dog():
    #A simple attempt to model a dog.#

    def __init__(self, name, age):
        #Initialize name and age attributes.#
        print('A new instance of Dog has been created.')
        self.name = name
        self.age = age

    def bark(self):
        print("Woof! woof! I'm " + self.name + " and I'm "
              + str(self.age) + " years old")

my_dog = Dog('willie', 6)
your_dog = Dog('lucy', 3)

my_dog.bark()
your_dog.bark()

print(my_dog.name + ' ' + str(my_dog.age))
print(your_dog.name + ' ' + str(your_dog.age))
```

Notar que el constructor de este nuevo programa imprime el mensaje `'A new instance of Dog has been created.'`, tal y como hacía el constructor del programa previo, pero las siguientes líneas de este método hacen algo nuevo. Ahora, el constructor también crea los atributos `name` y `age` dpara el nuevo

objeto instanciado, y les asigna los valores (pasados como argumentos desde el programa principal) de los parámetros `name` y `age`. De esta forma, en el programa principal, las líneas:

```
my_dog = Dog('willie', 6)
your_dog = Dog('lucy', 3)
```

, producen la creación de dos nuevas instancias de la clase `Dog`, la primera con los atributos `name` y `age` fijados a `'willie'` y 6, y la segunda con los atributos fijados a `'lucy'` y 3, respectivamente. Estos dos nuevos objetos se asignan a las variables `my_dog` y `your_dog`.

Para entender cómo funciona todo esto, vamos a explicar por fin cuál es el papel que juega el misterioso parámetro `self`. Como primer parámetro de cada método, `self` recibe automáticamente una referencia al objeto que está llamando a ese método. Esto significa que, gracias al parámetro `self`, el método puede acceder al objeto que lo ha llamado, y por consiguiente, a los atributos y el resto de métodos de ese objeto (e incluso crear nuevos atributos para ese objeto).

De vuelta al método constructor, y al ejecutarse la instrucción `my_dog = Dog('willie', 6)` desde el programa principal, el parámetro `self` recibe automáticamente una referencia a la nueva instancia `Dog` recién creada, mientras que los parámetros `name` y `age` reciben los argumentos `'willie'` y 6, respectivamente. Por lo tanto, las líneas:

```
self.name = name
self.age = age
```

, del método constructor crean los atributos `name` y `age` para la nueva instancia, y les asignan los valores de los parámetros `name` y `age`, que son `'willie'` y 6.

En el programa principal, la sentencia de asignación `my_dog = Dog('willie', 6)` asigna este nuevo objeto a la variable `my_dog`. Esto significa que `my_dog` se refiere ahora a una nueva instancia con su atributo `name` fijado a `'willie'` y su atributo `age` fijado a 6.

Por su parte, y a continuación, la siguiente instrucción:

```
your_dog = Dog('lucy', 3)
```

, desencadenará básicamente la misma secuencia de eventos, solo que esta vez creará una segunda instancia de la clase `Dog`, esta vez con los atributos `name` y `age` fijados a `'lucy'` y 3, respectivamente. Y este nuevo objeto se asigna a la variable `your_dog`.

ACCEDER A LOS ATRIBUTOS.

Evidentemente, los atributos son útiles porque podemos acceder a ellos y usarlos. Para mostrar cómo hacerlo, hemos modificado el método `bark()`. Ahora, cuando un perro ladra, además se presenta diciendo su nombre y su edad.

En primer lugar, hacemos que nuestro primer perro ladre llamando a su método `bark()` mediante la sentencia:

```
my_dog.bark()
```

A continuación, el método `bark()` recibe, a través del parámetro `self`, la referencia al objeto `Dog` que acaba de crearse:

```
def bark(self):
```

Después, `bark()` imprime el texto `Woof! woof! I'm willie and I'm 6 years old`, accediendo a los atributos `name` y `age` del objeto a través de `self.name` y `self.age`:

```
print("Woof! woof! I'm " + self.name + " and I'm "
+ str(self.age) + " years old")
```

Exactamente lo mismo ocurre cuando llamamos al método `bark()` para nuestra segunda instancia:

```
your_dog.bark()
```

, pero esta vez el método mostrará el texto `Woof! woof! I'm lucy and I'm 3 years old`, ya que ahora los atributos `name` y `age` contienen los valores `'lucy'` y 3, respectivamente.

Por supuesto, también podemos acceder y modificar los atributos de una instancia desde fuera de su clase. Por ejemplo, en el programa principal hemos accedido a los atributos `name` y `age` de los objetos `my_dog` y `your_dog` para imprimir sus valores:

```
print(my_dog.name + ' ' + str(my_dog.age))
print(your_dog.name + ' ' + str(your_dog.age))
```

IMPRIMIR TODO UN OBJETO.

Normalmente, si quisiéramos imprimir todo un objeto mediante la instrucción `print(my_dog)`, Python devolvería una salida parecida a la siguiente:

```
<__main__.Dog object at 0x022B18B0>
```

Este mensaje nos dice que hemos imprimido una instancia de `Dog` en el programa principal, pero no nos da ningún tipo de información útil acerca de esa instancia. Sin embargo, hay una forma de evitar esta contrariedad. Escribe lo siguiente en el editor de archivo, y guarda el programa como `dog4.py`:

```
class Dog():
    #A simple attempt to model a dog.#

    def __init__(self, name, age):
        #Initialize name and age attributes.#
        print('A new instance of Dog has been created.')
        self.name = name
        self.age = age

    def __str__(self):
        rep = "\nDog object\n"
        rep += "name: " + self.name + "\n"
        rep += "age: " + str(self.age)
        return rep

    def bark(self):
        print("Woof! woof! I'm " + self.name + " and I'm "
              + str(self.age) + " years old")
```

```
my_dog = Dog('willie', 6)
your_dog = Dog('lucy', 3)

print(my_dog)
print (your_dog)
```

El método especial `__str__()` que hemos incluido en la definición de la clase nos permite crear una representación tipo cadena de nuestros objetos, que se mostrará siempre que queramos imprimir un objeto

La cadena que devuelva este método será la cadena que se imprima para ese objeto. En nuestro caso, hemos decidido que el método devuelva una cadena que muestre los valores de los atributos de la instancia creada. Así, y por ejemplo, al ejecutar la instrucción:

```
print(my_dog)
```

, el texto que se muestra es:

```
Dog object
name: willie
age: 6
```

9.3. TRABAJANDO CON CLASES Y OBJETOS.

Podemos usar clases para representar muchas situaciones del mundo real. Una vez hayamos escrito una clase, invertiremos la mayor parte del resto del tiempo trabajando con objetos instanciados a partir de esa clase. Una de las primeras tareas a realizar será modificar los atributos asociados a una instancia particular. Podemos modificar los atributos de un objeto directamente, o escribir métodos que actualicen los atributos de formas específicas.

A modo de ejemplo, vamos a escribir una clase que represente un coche. Nuestra clase almacenará información acerca del tipo de coche con el que estamos trabajando, y tendrá un método para poder imprimir el objeto instanciado y mostrar esta información. Escribe lo siguiente en un editor de archivos, y guárdalo como `car1.py`:

```
class Car():
    """A simple attempt to represent a car."""

❶   def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year

❷   def __str__(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name

❸ my_new_car = Car('audi', 'a4', 2016)
  print(my_new_car)
```

En (1), dentro de la clase `Car`, definimos el método `__init()__` con el parámetro `self`, de la misma forma que lo hicimos en las secciones previas. A `__init()__` le damos tres parámetros: `make`, `model`, y `year`. El método `__init()__` recibe estos tres parámetros y se los asigna a los atributos que estarán

asociados a las instancias creadas a partir de esta clase. Cuando creemos un nuevo objeto `Car`, deberemos especificar un fabricante (`make`), un modelo (`model`), y un año (`year`) para esa instancia.

En (2) definimos el método `__str()__` que nos permitirá imprimir la instancia de un coche, mostrando sus atributos en la forma de una cadena adecuadamente formateada.

En (3) instanciamos un objeto de la clase `Car` y lo almacenamos en la variable `my_new_car`. A continuación imprimimos esta instancia para mostrar el tipo de coche creado:

```
2016 audi a4
```

Ahora, y para hacer esta clase más interesante, vamos a añadir un atributo que cambie a lo largo del tiempo: el kilometraje general del coche.

ASIGNAR UN VALOR POR DEFECTO A UN ATRIBUTO.

Todo atributo de una clase necesita un valor inicial, aunque sea el valor `0` o una cadena vacía. En muchos casos, el lugar adecuado para especificar el valor inicial de un atributo es en el método `__init()__`. Cuando hagamos esto, no será necesario pasar un argumento para darle un valor a ese atributo.

Vamos a añadir un atributo llamado `odometer_reading` cuyo valor inicial sea `0`. También añadiremos un método `read_odometer()` para leer el odómetro de cada instancia. Escribe lo siguiente en el editor de archivos, y guárdalo como `car2.py`:

```
class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
❶       self.odometer_reading = 0

    def __str__(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name

❷   def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " km on it.")

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car)
my_new_car.read_odometer()
```

Esta vez, cuando Python llame al método `__init()__` para crear una nueva instancia, almacenará los valores de los parámetros `make`, `model`, y `year` como atributos de la instancia. (1) A continuación, Python crea un nuevo atributo llamado `odometer_reading`, y fija su valor inicial a `0`. También hemos escrito un nuevo método, `read_odometer()`, que realiza la lectura del kilometraje actual en el odómetro del coche. Nuestro coche comienza con un kilometraje inicial de cero:

```
2016 audi a4
This car has 0 km on it.
```

MODIFICAR LOS VALORES DE UN ATRIBUTO.

Ahora, podemos cambiar el valor de un atributo de tres formas: (a) podemos cambiarlo directamente mediante una instancia, (b) podemos fijar su valor mediante un método, o (c) podemos incrementar su valor (esto es, sumarle o restarle una cierta cantidad) mediante un método.

Modificar el valor de un atributo directamente.

La forma más sencilla de modificar el valor de un atributo es hacerlo directamente mediante una instancia. En este ejemplo, fijamos la lectura del odómetro a 23 directamente en el programa principal:

```
class Car():
    """A simple attempt to represent a car."""


my_new_car = Car('audi', 'a4', 2016)
print(my_new_car)

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

En la línea `my_new_car.odometer_reading = 23` usamos la notación basada en puntos para acceder al atributo `odometer_reading` de la instancia `my_new_car` y fijar su valor directamente. Al ejecutar el programa, obtenemos la siguiente salida:

```
2016 audi a4
This car has 23 km on it.
```

A veces querremos acceder a un atributo directamente como lo hemos hecho aquí, pero otras veces querremos escribir un método que actualice el valor del atributo.

Modificar el valor de un atributo mediante un método.

A veces es útil disponer de métodos que actualicen ciertos atributos de nuestras instancias. En lugar de acceder al atributo directamente, podemos pasarle el nuevo valor a un método, y que sea él quien se encargue de actualizarlo internamente. He aquí un ejemplo:

```
class Car():
    """A simple attempt to represent a car."""

    def update_odometer(self, mileage):
        """Set the odometer reading to the given value."""
        self.odometer_reading = mileage


my_new_car = Car('audi', 'a4', 2016)
print(my_new_car)

my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

La única modificación del programa ha sido la adición del método `update_odometer()`. Este método recibe un kilometraje (parámetro `mileage`) y lo almacena en `self.odometer_reading`. En el programa principal, llamamos al método `update_odometer()` pasándole un 23 como argumento (que se registrará

en el parámetro `mileage` en la definición del método). Esto fijará la lectura del odómetro a 23. Al llamar al método `read_odometer`, Python imprime:

```
2016 Audi A4
This car has 23 km on it.
```

Podemos ampliar el método `update_odometer()` para que haga más tareas cada vez que se modifique la lectura del odómetro. A modo de ejemplo, vamos a añdirle una pequeña lógica para asegurarnos de que nadie intente disminuir el valor del kilometraje:

```
class Car():
    """A simple attempt to represent a car."""

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " km on it.")

    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")


my_new_car = Car('audi', 'a4', 2016)
print(my_new_car)

my_new_car.update_odometer(2540)
my_new_car.read_odometer()

my_new_car.update_odometer(30)
my_new_car.read_odometer()
```

Ahora, `update_odometer()` comprueba si el nuevo kilometraje tiene sentido antes de modificar el atributo. Si el nuevo kilometraje, `mileage`, es mayor o igual que el kilometraje ya existente, `self.odometer_reading`, el método permite actualizar la lectura del odómetro al nuevo kilometraje. En cambio, si el nuevo kilometraje es menor que el actual, mostramos una advertencia según la cual no se permite hace retroceder un odómetro.

Incrementar el valor de un atributo mediante un método.

En otras ocasiones no queremos darle un nuevo valor a un atributo, sino incrementar su valor en una cierta cantidad. Imaginar que nos compramos un coche usado y le hacemos 100 kilómetros entre el día que lo compramos y el día que lo registramos. He aquí un método que nos permite pasarle esta cantidad incremental y añadir ese valor a la lectura del odómetro del coche usado:

```
class Car():
    """A simple attempt to represent a car."""

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(self.odometer_reading) + " km on it.")
```

```
    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

❶   def increment_odometer(self, kilometers):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += kilometers


❷ my_used_car = Car('subaru', 'outback', 2013)
  print(my_used_car)

❸ my_used_car.update_odometer(23500)
  my_used_car.read_odometer()

❹ my_used_car.increment_odometer(100)
  my_used_car.read_odometer()
```

El nuevo método `increment_odometer()` en (1) recibe un número de kilómetros, y le suma este valor a `self.odometer_reading`. En (2) creamos un coche usado, `my_used_car`. En (3) fijamos el valor de su odómetro a $23500\,km$, llamando al método `update_odometer()` y pasándole como argumento 23500. Finalmente, en (4) llamamos al método `increment_odometer()` y le pasamos un 100 como argumento, para que le sume $100\,km$ a la lectura del odómetro entre el momento en que compramos el coche y el momento en el que lo registramos. La salida del programa es:

```
2013 subaru outback
This car has 23500 km on it.
This car has 23600 km on it.
```

A modo de ejercicio, podemos modificar este método para que rechace incrementos negativos, de forma que nadie use esta función para reducir la lectura de un odómetro.

9.4. ATRIBUTOS DE CLASE Y MÉTODOS ESTÁTICOS

Gracias a los atributos, diferentes instancias de la misma clase pueden tener sus propias características individuales. Por ejemplo, podríamos tener 10 coches distintos, cada uno de ellos con sus propios fabricante, modelo, año, y lectura del odómetro. Pero en ocasiones necesitaremos tener cierta información que describa a la clase en su conjunto, y no a las instancias individuales. Por ejemplo, puede que queramos contabilizar el número total de coches creados. Para ello, deberíamos darle a cada instancia de `Car` un atributo llamado `total`. Pero en ese caso, cada vez que instanciásemos un nuevo coche, deberíamos actualizar el valor de atributo `total` de *todas* las instancias ya creadas. Esto podría convertirse en una tarea ciertamente engorrosa. Afortunadamente, Python ofrece una forma de crear un atributo que esté asociado a la clase en sí misma, y no a los objetos individuales. A estos atributos se les llama **atributos de clase**. Una vez definido un atributo de clase, solo habrá una copia de este atributo, independientemente del número de objetos que hayamos instanciado a partir de esa clase.

Es probable que también queramos un método que esté asociado a la clase en sí misma, y no a las instancias creadas a partir de ella. Para ello, Python también dispone de los **métodos estáticos**. Como los métodos estáticos están asociados a la clase, suelen utilizarse para trabajar con los *atributos de clase*.

A modo de ejemplo, vamos a escribir una clase `Critter` que defina una mascota virtual. Por el momento, esta mascota será muy simple: su única característica será su nombre, el cual estará disponible en el atributo `name`, y lo único que podrá hacer es hablar para saludar y presentarse, mediante el método `talk()`. Con todo lo que ya sabemos, es muy fácil construir una clase tan simple como ésta. Pero además, el programa define un atributo de clase que contabilice el número total de mascotas instanciadas, e incluye un método estático para mostrar este número. Escribe lo siguiente en el editor de archivos, y guárdalo como `critter1.py`:

```
class Critter():
    """A virtual pet"""
    total = 0

    @staticmethod
    def status():
        print("\nThe total number of critters is", Critter.total)

    def __init__(self, name):
        print("A new critter has been born!")
        self.name = name
        Critter.total += 1

    def __str__(self):
        rep = "Critter object\n"
        rep += "name: " + self.name + "\n"
        return rep

    def talk(self):
        print("Hi.  I'm", self.name, "\n")

# main

print("Accessing the class attribute Critter.total:", end=" ")
print(Critter.total)

print("\nCreating critters.")
crit1 = Critter("critter 1")
crit1.talk()
crit2 = Critter("critter 2")
crit2.talk()
crit3 = Critter("critter 3")
crit3.talk()

Critter.status()

print("\nAccessing the class attribute through an object:", end= " ")
print(crit1.total)
```

CREAR UN ATRIBUTO DE CLASE.

La segunda línea de la definición de la clase `Critter`, esto es:

```
total = 0
```

, crea el atributo de clase `total` y le asigna el valor `0`. Cualquier asignación de este tipo, esto es, un valor asignado a una nueva variable fuera de un método, sirve para crear un atributo de clase. Esta sentencia de asignación solo se ejecuta una única vez, cuando Python lee la definición de la clase. Esto significa que el

atributo de clase existe incluso antes de crear una primera instancia de la clase. Esto implica que podemos usar un atributo de clase aunque no hayamos creado ningún objeto de la clase.

ACCEDER A UN ATRIBUTO DE CLASE.

Acceder a un atributo de clase es bien sencillo. En el programa hemos accedido a este atributo en varios lugares. Por ejemplo, en la línea 2 del programa principal hemos imprimido su valor mediante:

```
print (Critter.total)
```

En el método estático `status()`, también hemos imprimido su valor mediante:

```
print("\nThe total number of critters is", Critter.total)
```

En el método constructor, hemos incrementado el valor de este atributo de clase mediante la línea:

```
Critter.total += 1
```

, y como resultado, cada vez que se instancia un nuevo objeto `Critter`, el valor de este atributo aumenta en 1 unidad.

En general, para acceder a un atributo de clase, se usa la notación basada en puntos: Escribimos el nombre de la clase, seguido de un punto, seguido del nombre del atributo de clase.

Finalmente, también podemos acceder a un atributo de clase a través de una instancia de esa clase. Eso es exactamente lo que hicimos en la última línea del programa principal con la instrucción:

```
print(crit1.total)
```

Este código imprime el valor del atributo de clase `total` (y no de un atributo `total` asociado a esa instancia). Esto significa que también podemos leer el valor de un atributo de clase a través de cualquier objeto creado a partir de esa clase. Por consiguiente, podríamos haber usado `print(crit1.total)`, `print(crit2.total)`, `print(crit3.total)`, o `print(Critter.total)` al final del programa principal, y obtener exactamente el mismo resultado.

Pero cuidado: Aunque podemos utilizar un objeto para acceder a un atributo de clase, no podemos usar un objeto para asignar un nuevo valor a un atributo de clase. Si queremos cambiar el valor de un atributo de clase, debemos acceder a él a través de su nombre de clase.

CREAR UN MÉTODO ESTÁTICO.

El primer método de la clase `Critter` es el método `status()`:

```
def status():
     print("\nThe total number of critters is", Critter.total)
```

Notar que esta definición no tiene a `self` en su conjunto de parámetros. Esto es lógico, ya que este método está diseñado para ser llamado a través de una clase, y no a través de una instancia. Por consiguiente, a este método nunca se le pasará una referencia a la instancia, y no necesitará ese parámetro. Por supuesto, los métodos estáticos pueden recibir argumentos y tener parámetros, pero no necesitan este parámetro en concreto.

Aunque esta definición crea un método llamado `status()`, también hemos añadido un **decorador** (esto es, un elemento que modifica una función o un método) justo antes de la definición. Este decorador es el que permite crear un método estático con el mismo nombre:

```
@staticmethod
```

Como resultado, la clase tendrá un método estático, llamado `status()`, que muestra el número total de objetos `Critter`, imprimiendo por pantalla el valor del atributo de clase `total`.

En definitiva, para crear un método estático, debemos proceder como en este ejemplo: Comenzamos con el decorador `@staticmethod`, seguido de la definición del método de clase. Y como el método está definido para la clase en sí misma, no debemos incluirle el parámetro `self`, que solo es necesario para los métodos de instancia (o métodos de objeto).

LLAMAR A UN MÉTODO ESTÁTICO.

Llamar a un método estático es muy sencillo: Basta con escribir:

```
Critter.status()
```

Si invocásemos a este método en la primera línea del programa principal, este método mostraría por pantalla un `0`, ya que todavía no se han instanciado ningún objeto `Critter`. Pero notar que podemos llamar a este método aun cuando no se ha creado ni un solo objeto. Como los métodos estáticos se pueden llamar mediante las clases, no se necesitan instancias para poder invocarlos.

A continuación, creamos tres objetos `Critter` en el programa principal. Si a continuación volviésemos a llamar al método `status()`, se imprimiría un `3` por pantalla, ya que ahora hay tres objetos de la clase `Critter`. Esto funciona bien porque, durante la ejecución del método constructor para cada instancia, el atributo de clase `total` se incrementa en 1 unidad.

9.5. ATRIBUTOS Y MÉTODOS PRIVADOS.

Por defecto, todos los atributos y métodos de un objeto son públicos, lo que significa que se puede acceder a ellos desde cualquier otra parte de nuestro programa. Sin embargo, también podemos definir atributos y métodos que sean privados, lo que significa que no serán visibles fuera del objeto, esto es, que solo los métodos del propio objeto podrán acceder a ellos. A modo de ejemplo, escribe el siguiente programa en el editor de archivos, y guárdalo como `critter 2.py`:

```
class Critter():
    """A virtual pet"""
    def __init__(self, name, mood):
        print("A new critter has been born!")
        self.name = name            # public attribute
        self.__mood = mood          # private attribute

    def talk(self):
        print("\nI'm", self.name)
        print("Right now I feel", self.__mood, "\n")

    def __private_method(self):
        print("This is a private method.")

    def public_method(self):
        print("This is a public method.")
        self.__private_method()
```

```
# main
crit = Critter(name = "Poochie", mood = "happy")
crit.talk()
crit.public_method()
```

CREAR ATRIBUTOS PRIVADOS.

Para limitar el acceso directo a ciertos atributos de un objeto desde un código externo al objeto podemos usar **atributos privados**. En este programa, hemos creado dos atributos, uno público y otro privado:

```
class Critter():
    """A virtual pet"""
    def __init__(self, name, mood):
        print("A new critter has been born!")
        self.name = name            # public attribute
        self.__mood = mood          # private attribute
```

Los dos guiones bajos con los que empieza el nombre del segundo atributo le dicen a Python que se trata de un atributo privado. Por lo tanto, para crear nuestros propios atributos privados, basta con comenzar el nombre del atributo con dos guiones bajos.

ACCEDER A LOS ATRIBUTOS PRIVADOS.

Es perfectamente legal acceder a un atributo privado de un objeto dentro de la definición de la clase de ese objeto. (Recordemos que los atributos privados existen para evitar que el código externo a una clase pueda acceder directamente a ese atributo). En el programa ejemplo hemos accedido al atributo privado dentro del método `talk()`:

```
    def talk(self):
        print("\nI'm", self.name)
        print("Right now I feel", self.__mood, "\n")
```

Este método imprime el valor del atributo privado del objeto, que representa el estado de ánimo (mood) de la mascota.

Si intentásemos acceder a este atributo desde fuera de la definición de la clase `Critter`, obtendríamos un mensaje de error. Escribe lo siguiente en el programa principal y observa la salida que genera Python:

```
crit = Critter(name = "Poochie", mood = "happy")
print (crit.mood)
```

```
A new critter has been born!
Traceback (most recent call last):
  File "C:\Users\Usuario\Dropbox\PYTHON\PROGRAMAS\8 critter 2.py",
line 22, in <module>
    print (crit.mood)
AttributeError: 'Critter' object has no attribute 'mood'
```

Al lanzar una excepción `AtributeError`, Python nos está diciendo que `crit` no tiene un atributo `mood`. Por supuesto, tampoco funciona el acceso a este atributo añadiendo los dos guiones bajos. Escribe lo siguiente en el programa principal y observa la salida que produce Python:

```
crit = Critter(name = "Poochie", mood = "happy")
print (crit.__mood)
```

```
A new critter has been born!
Traceback (most recent call last):
  File "C:\Users\Usuario\Dropbox\PYTHON\PROGRAMAS\8 critter 2.py",
line 22, in <module>
    print (crit.__mood)
AttributeError: 'Critter' object has no attribute '__mood'
```

Este intento de "engañar" a Python también ha producido un mensaje de error. De nuevo, Python nos está diciendo el atributo `__mood` no existe. ¿Significa esto que el valor de un atributo privado es totalmente inaccesible desde fuera de la definición de su clase? Bueno, realmente no. Python únicamente oculta el atributo mediante una convención especial de nombres, pero todavía es técnicamente posible acceder a un atributo privado. Esto es precisamente lo que vamos a hacer a continuación. Escribe lo siguiente en el programa principal y observa la salida que general Python:

```
crit = Critter(name = "Poochie", mood = "happy")
print(crit._Critter__mood)
```

```
A new critter has been born!
happy
```

La instrucción `print(crit._Critter__mood)` imprime el elusivo valor del atributo privado, que en nuestro caso en la cadena `"happy"`.

Como también es posible acceder a los atributos privados, puede que nos preguntemos para sentido tiene la existencia de tales atributos. Bueno, en Python, el hecho de indicar que un atributo es privado es una forma de decir que ese atributo es solo para uso interno. Además, nos ayuda a evitar accesos no intencionados a esos atributos o métodos.

CREAR MÉTODOS PRIVADOS.

Podemos crear un método privado exactamente de la misma forma que creamos un atributo privado: Basta con añadir dos guiones bajos al comienzo de su nombre. Eso es lo que hemos hecho en la definición del método privado `__private_method()` en la clase de nuestro ejemplo:

```
    def __private_method(self):
        print("This is a private method.")
```

Aunque se trata de un método privado, también puede ser accedido por cualquier otro método de la clase. Al igual que los atributos privados, los métodos privados están pensados para ser accedidos únicamente a través de los métodos de la propia clase.

ACCEDER A LOS MÉTODOS PRIVADOS.

Como con los atributos privados, el acceso a los métodos privados desde dentro de la definición de la clase es una tarea sencilla. En el método `public_method()` de nuestro programa ejemplo hemos accedido al método privado de la clase:

```
    def public_method(self):
        print("This is a public method.")
        self.__private_method()
```

Este método imprime la cadena `"This is a public method."`, y a continuación llama al método privado, que imprime la cadena `"This is a private method."`,

Al igual que en los atributos privados, los métodos privados no están pensados para ser accedidos desde el código fuera de la definición de la clase. Escribe lo siguiente en el programa principal para intentar acceder al método privado de la instancia `crit`, y observa la salida de Python:

```
crit = Critter(name = "Poochie", mood = "happy")
crit.private_method()
```

```
A new critter has been born!
Traceback (most recent call last):
  File "C:\Users\Usuario\Dropbox\PYTHON\PROGRAMAS\8 critter 2.py",
line 22, in <module>
    crit.private_method()
AttributeError: 'Critter' object has no attribute 'private_method'
```

Este código lanza la familiar excepción `AtributeError`, mediante la cual Python informa de que no existe un método con este nombre. Python oculta este método mediante la convención de nombres basada en el doble guión bajo. Por supuesto, si intentamos acceder al método añadiendo al nombre los dos guiones bajos, Python genera la misma excepción:

```
crit = Critter(name = "Poochie", mood = "happy")
crit.__private_method()
```

```
A new critter has been born!
Traceback (most recent call last):
  File "C:\Users\Usuario\Dropbox\PYTHON\PROGRAMAS\8 critter 2.py", line 22
, in <module>
    crit.__private_method()
AttributeError: 'Critter' object has no attribute '__private_method'
```

Sin embargo, y al igual que con los atributos privados, es técnicamente posible acceder a los métodos privados desde cualquier parte del programa:

```
crit = Critter(name = "Poochie", mood = "happy")
crit._Critter__private_method()
```

```
A new critter has been born!
This is a private method.
```

Pero como ya hemos enfatizado varias veces, nunca deberíamos acceder a los métodos privados de una instancia desde el código externo a su definición de clase.

RESPETAR LA PRIVACIDAD DE UN OBJETO.

En el programa principal de nuestro programa ejemplo hemos respetado las normas de buena conducta, y no hemos intentado acceder a los atributos y métodos privados de la clase. Para ello, hemos creado un objeto y hemos llamado a sus dos métodos públicos:

```
# main

crit = Critter(name = "Poochie", mood = "happy")
crit.talk()
crit.public_method()
```

El método `__init__()` del objeto, al que se llama automáticamente justo después de que se haya creado el objeto, le dice al mundo que ha nacido una nueva mascota (imprimiendo la cadena "`A new critter`

has been born!"). El método talk() del objeto crit nos dice cómo se siente esa mascota. El método public_method() imprime la cadena "This is a public method.", y a continuación, llama al método privado de crit, que imprime la cadena "This is a private method.". Finalmente, el programa termina.

ENTENDER CUÁNDO IMPLEMENTAR LA PRIVACIDAD.

Ahora que ya sabemos cómo usar la privacidad, ¿crees que es conveniente establecer como privados todos los atributos de nuestras clases, para protegerlos del mundo exterior? Por supuesto que no. Sólo debemos hacer privado todo aquel método o atributo que no queramos que sea accedido por un código ajeno a la definición de clase. La regla general entre los programadores de Python es asegurar que el código externo solo usará los métodos de un objeto y no alterará directamente los atributos de ese objeto. Algunos consejos que siempre debemos tener en mente son los siguientes:

Al escribir una clase debemos:
- Crear métodos para reducir la necesidad del código externo de acceder directamente a los atributos de un objeto.
- Usar privacidad para aquellos atributos y métodos que son exclusivamente internos a la funcionalidad y operatividad de los objetos.

Al usar un objeto debemos:
- Minimizar la lectura directa de los atributos de ese objeto.
- Evitar alterar directamente los atributos del objeto.
- No intentar nunca acceder directamente a los atributos o métodos privados de un objeto.

9.6. CONTROLAR EL ACCESO A LOS ATRIBUTOS: PROPIEDADES.

En vez impedir por completo el acceso a un atributo privado, en ocasiones solo necesitaremos *limitar* el acceso a ese atributo. Por ejemplo, tal vez tengamos un atributo privado que queremos que pueda leerlo el código externo a la clase, pero que no pueda cambiarlo. Python proporciona unas cuantas herramientas para permitir este tipo de acciones: Las **propiedades** nos permiten gestionar la forma en la que podemos acceder o cambiar un atributo.

A modo de ejemplo, vamos a escribir un programa para nuestra mascota virtual en el que el código externo pueda leer los atributos de los objetos de la clase Critter, pero que impone restricciones al código externo cuando intenta cambiar su valor. En concreto, si ese código intenta asignar una cadena vacía al atributo, el programa se queja y no lo permite. Escribe lo siguiente en una nueva ventana del editor de archivos, y guarda el programa como critter 3.py:

```
class Critter():
    """A virtual pet"""
    def __init__(self, name):
        print("A new critter has been born!")
        self.__name = name

    @property
    def name(self):
        return self.__name
```

```
    @name.setter
    def name(self, new_name):
        if new_name == "":
            print("A critter's name can't be the empty string.")
        else:
            self.__name = new_name
            print("Name change successful.")

    def talk(self):
        print("\nHi, I'm", self.name)

# main
crit = Critter("Poochie")
crit.talk()

print("\nMy critter's name is:", end= " ")
print(crit.name)

print("\nAttempting to change my critter's name to Randolph...")
crit.name = "Randolph"
print("My critter's name is:", end= " ")
print(crit.name)

print("\nAttempting to change my critter's name to the empty strin
crit.name = ""
print("My critter's name is:", end= " ")
print(crit.name)
```

CREAR PROPIEDADES.

Una forma de controlar el acceso a un atributo privado es crear una *propiedad*, esto es, un objeto con métodos que permiten el acceso indirecto a sus atributos, y que a menudo imponen algún tipo de restricción a ese acceso. Después del constructor de la clase `Critter`, hemos creado una propiedad llamada `name` para permitir el acceso inderecto al atributo privado `__name`:

```
    @property
    def name(self):
        return self.__name
```

Aquí hemos creado una propiedad escribiendo un método que devuelve el valor del atributo privado al que queremos proporcionarle acceso indirecto (en este caso, `__name`), y precediendo la definición del método con el decorador `@property`. La propiedad tiene el mismo nombre que el método, en este caso, `name`. A partir de este momento podremos usar la propiedad `name` de cualquier instancia `Critter` para obtener (usando la notación basada en puntos) el valor del atributo privado `__name`, ya sea desde dentro o desde fuera de la definición de la clase. Veremos ejemplos de cómo usar esta propiedad en la siguiente sección.

Para crear propiedades en nuestro propios programas, el procedimiento consiste en escribir un método que devuelva el valor al que queremos proporcionar acceso indirecto, y preceder la definición del método con el decorador `@property`. La propiedad tendrá el mismo nombre que el método. Si estamos proporcionando acceso a un atributo privado, la convención en Python es darle a la propiedad el mismo nombre que el atributo privado sin los guiones bajos, tal y como hemos hecho aquí.

En el ejemplo previo, al crear la propiedad `name` hemos proporcionado acceso de solo lectura al atributo privado `__name`. Sin embargo, una propiedad también puede proporcionar acceso de escritura, o incluso imponer algunos límites a ese acceso.

En el programa `critter 3.py` también hemos proporcionado acceso de lectura, con algunos límites, al atributo privado `__name` a través de la propiedad `name`:

```
    @name.setter
    def name(self, new_name):
        if new_name == "":
            print("A critter's name can't be the empty string.")
        else:
            self.__name = new_name
            print("Name change successful.")
```

Comenzamos el código con el decorador `@name.setter`. Al acceder al atributo `setter` de la propiedad `name`, lo que estamos diciendo es que la siguiente definición del método proporcionará una forma de establecer (set) el valor de la propiedad `name`. Para crear nuestros propios decoradores para establecer el nombre de una propiedad, basta con seguir este ejemplo: Comenzamos con el símbolo `@`, seguido del nombre de la propiedad, seguido de un punto (`.`), seguido de la palabra reservada `setter`.

Después del decorador, definimos un método llamado `name` al que llama el código externo que está intentando darle un nuevo valor al atributo privado, usando la propiedad recién definida. Notar que este método se llama `name` igual que la propiedad, tal y como debe ser. El método "establecedor" (setter) debe tener el mismo nombre que la propiedad correspondiente.

Dentro del método, el parámetro `new_name` recibe el valor del nuevo nombre de la mascota. Si el valor es una cadena vacía, `__name` permanece inalterado, y se muestra un mensaje que indica que este cambio de nombre no es posible. En caso contrario, el método fija el valor del atributo privado `__name` al valor del parámetro `new_name`, y muestra un mensaje que informa de que el cambio se ha producido. Y como hemos hecho en este ejemplo, cuando construyamos un método para fijar el valor de una propiedad, debemos recordar que la definición del método debe tener un parámetro para recibir el nuevo valor de la propiedad.

ACCEDER A LAS PROPIEDADES.

Tras crear la propiedad `name`, podemos obtener el nombre de un objeto `Critter` mediante la notación basada en puntos. La siguiente parte del programa muestra cómo hacerlo

```
    def talk(self):
        print("\nHi, I'm", self.name)

# main
crit = Critter("Poochie")
crit.talk()
```

La instrucción `self.name` accede a la propiedad `name`, y llama indirectamente al método que devuelve `__name`. En este caso, el método devolvería la cadena `"Poochie"`. Pero no solo podemos usar la propiedad `name` de una instancia dentro de la definición de su clase, también podemos usarla fuera, como hacemos a continuación:

```
print("\nMy critter's name is:", end= " ")
print(crit.name)
```

Aunque este código está fuera de la definición de la clase `Critter`, hace que ocurran esencialmente las mismas cosas: La instrucción `crit.name` accede a la propiedad `name` del objeto `Critter`, y llama indirectamente al método que devuelve `__name`. De nuevo, el método devolvería la cadena `"Poochie"`.

A continuación, vamos a tratar de cambiar el nombre de la mascota:

```
print("\nAttempting to change my critter's name to Randolph...")
crit.name = "Randolph"
print("My critter's name is:", end= " ")
print(crit.name)
```

La sentencia de asignación `crit.name = "Randolph"` accede a la propiedad `name` del objeto, y llama indirectamente al método que intenta fijar el valor de `__name`. En este caso, se le pasa la cadena `"Randolph"` al parámetro `new_name` del método, y como `"Randolph"` no es una cadena vacía, el atributo `__name` del objeto se cambia a `"Randolph"`, y se muestra el mensaje `Name change successful`. Si ahora mostramos el nombre del objeto usando la propiedad `name` mediante la instrucción `print(crit.name)`, el programa indicará que el nombre de nuestro objeto es `"Randolph"`.

Para terminar, intentamos cambiar el nombre del objeto a una cadena vacía:

```
print("\nAttempting to change my critter's name to the empty string...")
crit.name = ""
print("My critter's name is:", end= " ")
print(crit.name)
```

Como antes, la sentencia de asignación accede a la propiedad `name` del objeto, y llama indirectamente al método que intenta fijar el valor de `__name` a una cadena vacía. En este caso, se le pasa una cadena vacía al parámetro `new_name` del método, y como resultado, el método lanza el mensaje `A critter's name can't be the empty string`, y el atributo `__name` del objeto permanece inalterado. La instrucción `print(crit.name)` nos permite comprobar que el nombre del objeto sigue siendo `"Randolph"`, porque el nombre no se ha cambiado.

9.7. PROYECTO: MASCOTA VIRTUAL.

Vamos a escribir un programa en el que el usuario debe cuidar de una mascota virtual como las que hemos estado usando en las últimas secciones de este capítulo. El usuario le pondrá un nombre, y será responsable de su cuidado. Para ello, el usuario deberá alimentarla y jugar con ella para mantenerla de buen humor. El usuario podrá escuchar a su mascota, para saber cómo se siente. El humor de la mascota podrá variar de feliz a enojada. El programa incluirá un menú que le permitirá al usuario interactuar con su mascota:

```
I'm Poochie and I feel happy now.


        Critter Caretaker

        0 - Quit
        1 - Listen to your critter
        2 - Feed your critter
        3 - Play with your critter

Choice:
```

Aunque podríamos trabajar sin recurrir a objetos, vamos a crear nuestra mascota como un objeto. Esto nos permitirá que el programa sea más sencillo y flexible: Una vez hayamos construido la clase para nuestra mascota, será muy fácil crear una, dos, tres, o una docena de ellas. De hecho, uno de los ejercicios del capítulo es crear y gestionar toda una granja de mascotas.

LA CLASE CRITTER.

La clase `Critter` será la plantilla para el objeto que representará la mascota del usuario. Esta clase ya nos resultará familiar de secciones previas. No es una clase complicada, pero ocupa unas cuantas líneas de código, así que vale la pena construirla paso a paso. Abre una nueva ventana en el editor de archivo, y ve construyendo poco a poco el programa. Cuando lo guardes, llama al archivo `critter_caretaker.py`.

El método constructor.

El método constructor de la clase inicializa los tres atributos públicos de todo objeto `Critter`, a saber, `name` (nombre), `hunger` (apetito), y `boredom` (aburrimiento). Los atributos `hunger` y `boredom` tendrán unos valores por defecto de `0`, lo que permitirá que la mascota comience de muy buen humor:

```
class Critter():
    """A virtual pet"""
    def __init__(self, name, hunger = 0, boredom = 0):
        self.name = name
        self.hunger = hunger
        self.boredom = boredom
```

El método __pass_time().

El método `__pass_time()` es un método privado que incrementa los niveles de apetito y de aburrimiento de la mascota. A este método se le llama al final de cada método en el que la mascota hace algo (comer, jugar, o hablar) para simular el paso del tiempo. Hemos hecho este método privado porque solo debería ser llamado por otro método de la clase, y no desde el exterior. En este programa solo percibiremos el paso del tiempo cuando la mascota haga algo, como comer, jugar, o hablar.

```
def __pass_time(self):
    self.hunger += 1
    self.boredom += 1
```

La propiedad mood.

La propiedad `mood` representa el humor o estado de ánimo de la mascota. El siguiente método calcula su estado de ánimo. Para hacerlo, suma los valores de los atributos `hunger` y `boredom`, y basándose en el total, devuelve una cadena para el estado de ánimo: `"happy"`, `"okay"`, `"frustrated"`, o `"mad"`.

Lo interesante de la propiedad `mood` es que no proporciona acceso a ningún atributo privado. Esto es así porque la cadena que representa el estado de ánimo de la mascota no se guarda como parte del objeto `Critter`, sino que se calcula sobre la marcha. La propiedad `mood` simplemente proporciona acceso a la cadena devuelta por el método.

```
@property
def mood(self):
    unhappiness = self.hunger + self.boredom
    if unhappiness < 5:
        m = "happy"
    elif 5 <= unhappiness <= 10:
        m = "okay"
    elif 11 <= unhappiness <= 15:
        m = "frustrated"
    else:
        m = "mad"
    return m
```

El método talk().

El método `talk()` informa al mundo sobre el estado de ánimo de la mascota, accediendo a la propiedad `mood`. A continuación, llama al método `__pass_time()`.

```
def talk(self):
    print("I'm", self.name, "and I feel", self.mood, "now.\n")
    self.__pass_time()
```

El método eat().

El método `eat()` reduce el nivel de apetito en una cantidad que se le pasa como argumento (parámetro `food`). Si no se le pasa valor alguno, `food` toma el valor por defecto de 4. El nivel de apetito de la mascota siempre se está controlando, y nunca se permite que caiga por debajo de 0. Finalmente, el método también llama a `__pass_time()`.

```
def eat(self, food = 4):
    print("Brruppp.  Thank you.")
    self.hunger -= food
    if self.hunger < 0:
        self.hunger = 0
    self.__pass_time()
```

El método play().

El método `play()` reduce el nivel de aburrimiento de la mascota en una cantidad que se le pasa como argumento al parámetro `fun`. Si no se le pasa valor alguno, `fun` toma un valor por defecto de 4. El nivel de aburrimiento de la mascota siempre se está controlando, y no se permite que caiga por debajo de 0. Al final, el método llama a `__pass_time()`.

```
def play(self, fun = 4):
    print("Wheee!")
    self.boredom -= fun
    if self.boredom < 0:
        self.boredom = 0
    self.__pass_time()
```

CREAR LA MASCOTA.

En esta ocasión hemos decidido poner el programa principal dentro de una función, la función `main()`. Al comenzar el programa, le pedimos al usuario el nombre de la mascota. A continuación, instanciamos un nuevo objeto `Critter`. Como no hemos proporcionado valores para `hunger` y `boredom`, los atributos comienzan por defecto a 0, y la mascota empieza su vida contenta y feliz.

```
def main():
    crit_name = input("What do you want to name your critter?: ")
    crit = Critter(crit_name)
```

CREAR EL MENÚ.

A continuación, hemos creado el menú de interacción con la mascota. Si el usuario inserta un 0, el programa termina. Si el usuario inserta un 1, se llama al método `talk()` del objeto. Si el usuario introduce un 2, se llama al método `eat()` del objeto. Si el usuario inserta un 3, se llama al método `play()` del objeto. Finalmente, si el usuario introduce cualquier otra cosa, el programa le indica que su opción no es válida.

```
choice = None
while choice != "0":
    print \
    ("""
    Critter Caretaker

    0 - Quit
    1 - Listen to your critter
    2 - Feed your critter
    3 - Play with your critter
    """)

    choice = input("Choice: ")
    print()

    # exit
    if choice == "0":
        print("Good-bye.")

    # listen to your critter
    elif choice == "1":
        crit.talk()

    # feed your critter
    elif choice == "2":
        crit.eat()

    # play with your critter
    elif choice == "3":
        crit.play()

    # some unknown choice
    else:
        print("\nSorry, but", choice, "isn't a valid choice.")
```

EMPEZAR EL PROGRAMA.

Para empezar el programa, escribimos una sola línea de código que llame a la función `main()`:

```
main()
```

Y con esto, el programa ya está acabado. Pruébalo para comprobar que funciona correctamente, o para localizar posibles errores.

9.8. ENVÍO Y RECEPCIÓN DE MENSAJES.

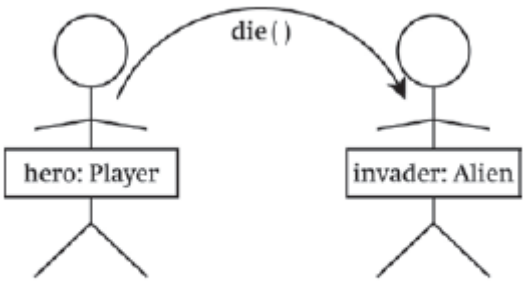
Los programas que hemos visto hasta ahora involucraban a un único objeto. Ésa es una buena forma de aprender a usar la POO, pero la verdadera potencia de la POO solo puede apreciarse haciendo que varios objetos trabajen juntos. A partir de esta sección y en adelante, aprenderemos a crear múltiples objetos y definir relaciones entre ellos, de forma que puedan interactuar. Para comenzar, veremos la forma en la que los objetos pueden enviar y recibir mensajes.

En cierto sentido, un programa orientado a objetos es como un ecosistema, y los objetos son como organismos. Y para mantener un ecosistema equilibrado, los organismos deben interactuar. Lo mismo ocurre con la POO. Para escribir programas útiles, los objetos deben interaccionar de formas bien definidas. En la jerga de la POO, los objetos interactúan *enviándose mensajes* entre ellos. A nivel práctico, lo que hacen los

objetos es llamar a los métodos de los otros objetos. Puede que esto parezca un poco invasivo, pero en todo caso, es mucho más respetuoso que acceder a los atributos de otro objeto.

A modo de ejemplo, vamos a programar un videojuego de acción en el que el jugador dispara a un alienígena. En el programa, el héroe le dispara a un invasor, y el invasor muere (no sin antes dar un largo discurso de despedida). El programa realizará esta tarea en el momento en el que un objeto le envíe un mensaje al otro objeto.

Técnicamente, lo que ocurre es que el programa instancia un objeto `hero` de la clase `Player`, y un objeto `invader` de la clase `Alien`. Cuando el programa llame al método `blast()` del objeto `hero` pasándole `invader` como argumento, `hero` llamará a su vez al método `die()` de `invader`. En palabras, cuando un jugador dispara a un alienígena, el jugador le envía un mensaje al alienígena para decirle que se muera.



hero.blast(invader)

A continuación mostramos el código del programa. Cópialo en una nueva ventana del editor de archivos y guárdalo como `alien_blaster.py`:

```
class Player(object):
    """ A player in a shooter game. """
    def blast(self, enemy):
        print("The player blasts an enemy.\n")
        enemy.die()

class Alien(object):
    """ An alien in a shooter game. """
    def die(self):
        print("The alien gasps and says, 'Oh, this is it." \
              "This is the big one. \n" \
              "Yes, it's getting dark now. \n" \
              "Tell my 1.6 million larvae that I loved them... \n" \
              "Good-bye, cruel universe.'")

# main
print("\t\tDeath of an Alien\n")

hero = Player()
invader = Alien()
hero.blast(invader)
```

ENVIAR UN MENSAJE.

Evidentemente, antes de que un objeto le pueda enviar a otro objeto un mensaje, necesitamos tener los dos objetos. Por eso, hemos instanciado dos objetos en el programa principal: el objeto `hero` de la clase `Player`, y el objeto `invader` de la clase `Alien`.

La siguiente línea de código es interesante: Mediante la instrucción `hero.blast(invader)`, llamamos al método `blast()` del objeto `hero` (de la clase `Player`) y le pasamos el objeto `invader` (de la clase

Alien) como argumento. Analizando la definición de blast(), podemos ver que el método almacena el objeto pasado como argumento en su parámetro enemy. Así, cuando blast() se ejecuta, enemy se refiere al objeto Alien. Después de mostrar un mensaje, blast() llama al método die() del objeto Alien, mediante la instrucción enemy.die(). Básicamente, lo que ocurre es que el objeto Player envía un mensaje al objeto Alien, y eso lo hace llamando a su método die().

RECIBIR UN MENSAJE.

El objeto Alien recibe el mensaje del objeto Player cuando éste último llama a su método die(). Entonces, el método die() del objeto Alien muestra por pantalla una melodramática despedida.

9.9. COMBINAR OBJETOS.

En el mundo real, los objetos más interesantes suelen estar compuestos por otros objetos independientes. Por ejemplo, un coche de carreras puede verse como un solo objeto compuesto de otros objetos individuales, como un chasis, cuatro neumáticos, y un motor. En otras ocasiones, puede que sea útil considerar que un objeto es una colección de otros objetos. Por ejemplo, un zoo puede verse como una colección de animales. Pues bien, en POO podemos imitar este tipo de relaciones para escribir nuestros programas. Así pues, podríamos escribir una clase Race_car que tenga un atributo llamado engine que se refiera a un objeto de la clase Race_engine. O podríamos escribir una clase Zoo que tenga un atributo animals que sea una lista de varios objetos de la clase Animal. Combinando objetos de esta forma podemos crear objetos complejos a partir de otros objetos más simples.

A modo de ejemplo, vamos a escribir un programa para un juego de cartas. Este programa usará objetos para representar las cartas individuales que podríamos tener en una mano de Blackjack o de Póker. El programa representará una mano de cartas mediante un objeto que sea una colección de objetos individuales de tipo carta. Escribe lo siguiente en el editor de archivos, y guarda el programa como playing_cards.py:

```
class Card():
    """ A playing card. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7",
             "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]

    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

    def __str__(self):
        rep = self.rank + self.suit
        return rep

class Hand():
    """ A hand of playing cards. """
    def __init__(self):
        self.cards = []

    def __str__(self):
        if self.cards:
            rep = ""
            for card in self.cards:
                rep += str(card) + "  "
        else:
            rep = "<empty>"
        return rep
```

```
    def clear(self):
        self.cards = []

    def add(self, card):
        self.cards.append(card)

    def give(self, card, other_hand):
        self.cards.remove(card)
        other_hand.add(card)

# main
card1 = Card(rank = "A", suit = "c")
print("Printing a Card object:")
print(card1)

card2 = Card(rank = "2", suit = "c")
card3 = Card(rank = "3", suit = "c")
card4 = Card(rank = "4", suit = "c")
card5 = Card(rank = "5", suit = "c")
print("\nPrinting the rest of the objects individually:")
print(card2)
print(card3)
print(card4)
print(card5)

my_hand = Hand()
print("\nPrinting my hand before I add any cards:")
print(my_hand)

my_hand.add(card1)
my_hand.add(card2)
my_hand.add(card3)
my_hand.add(card4)
my_hand.add(card5)
print("\nPrinting my hand after adding 5 cards:")
print(my_hand)

your_hand = Hand()
my_hand.give(card1, your_hand)
my_hand.give(card2, your_hand)
print("\nGave the first two cards from my hand to your hand.")
print("Your hand:")
print(your_hand)
print("My hand:")
print(my_hand)

my_hand.clear()
print("\nMy hand after clearing it:")
print(my_hand)
```

LA CLASE CARD.

Lo primero que hacemos en el programa es crear la clase `Card`, para instanciar objetos que representarán las cartas de nuestra mano.

Cada objeto `Card` tiene un atributo `rank` (valor), que representa el valor de la carta. Los posibles valores están listados en el atributo de clase `RANKS`: `"A"` representa un as, del `"2"` al `"10"` representan los valores numéricos correspondientes, `"J"` representa una sota (jack), `"Q"` representa una reina, y `"K"` representa un rey.

Cada carta también dispone de un atributo `suit` (palo), que representa el palo de la carta. Los posibles valores de este atributo están listados en el atributo de clase `SUITS`: `"c"` (club) representa tréboles, `"d"` indica diamantes, `"h"` significa corazones, y `"s"` (spades) representa picas. Así pues, un objeto con el valor del atributo `rank` igual a `"A"` y el valor del atributo `suit` igual a `"d"` representa el as de diamantes (`"Ad"`).

El método especial `__str__()` simplemente devuelve la concatenación de los atributos rank y suit, para que se pueda imprimir el objeto.

LA CLASE HAND.

Lo siguiente que hacemos en el programa es definir la clase `Hand`, para representar objetos que son una colección de objetos `Card`.

Cada nuevo objeto `Hand` tendrá un atributo `cards` que será una lista de objetos tipo `Card`, esto es, las cartas disponibles en la mano actual.

El método especial `__str()__` devuelve una cadena que representa la mano de cartas. Este método itera a través de cada objeto `Card` dentro del objeto `Hand` y concatena la representación en forma de cadena de cada objeto `Card`. Si el objeto `Hand` no dispone aún de ningún objeto `Card`, devuelve la cadena `"<empty>"`.

El método `clear()` limpia la lista de cartas asignando una lista vacía al atributo `cards`.

El método `add()` añade un nuevo objeto `Card` al atributo `cards` del objeto `Hand`.

El método `give()`quita un objeto `Card` de un objeto `Hand` y se lo da a otro objeto `Hand`, llamando al método `add()` de ese otro objeto. Otra forma de decir lo mismo es que el primer objeto `Hand` le envía un mensaje al segundo objeto `Hand` para añadirle uno de sus objetos `Card`.

USAR OBJETOS DE LA CLASE CARD.

En el programa principal, comenzamos creando e imprimiendo cinco objetos de tipo `Card`:

```
# main
card1 = Card(rank = "A", suit = "c")
print("Printing a Card object:")
print(card1)

card2 = Card(rank = "2", suit = "c")
card3 = Card(rank = "3", suit = "c")
card4 = Card(rank = "4", suit = "c")
card5 = Card(rank = "5", suit = "c")
print("\nPrinting the rest of the objects individually:")
print(card2)
print(card3)
print(card4)
print(card5)
```

El primer objeto `Card` se crea con un atributo `rank` igual a `"A"` y un atributo `suit` de `"c"`. Cuando imprimimos el objeto, se muestra por pantalla como `Ac`. El resto de objetos siguen la misma secuencia de acciones.

COMBINAR OBJETOS CARD USANDO UN OBJETO HAND.

A continuación, creamos un objeto `Hand`, lo asignamos a la variable `my_hand`, y lo imprimimos:

```
my_hand = Hand()
print("\nPrinting my hand before I add any cards:")
print(my_hand)
```

Como el atributo `cards` del nuevo objeto `Hand` es una lista vacía, al imprimir el objeto se mostrará la cadena `"<empty>"`.

Después, añadimos los cinco objetos `Card` creados antes a `my_hand`, y la imprimimos nuevamente:

```
my_hand.add(card1)
my_hand.add(card2)
my_hand.add(card3)
my_hand.add(card4)
my_hand.add(card5)
print("\nPrinting my hand after adding 5 cards:")
print(my_hand)
```

Esta vez, la impresión muestra el texto `Ac 2c 3c 4c 5c`.

Luego, creamos otro objeto `Hand` y lo asignamos a `your_hand`. Usando el método `give()` de `my_hand` transferimos las dos primeras cartas de `my_hand` a `your_hand`. Entonces, imprimimos las dos manos:

```
your_hand = Hand()
my_hand.give(card1, your_hand)
my_hand.give(card2, your_hand)
print("\nGave the first two cards from my hand to your hand.")
print("Your hand:")
print(your_hand)
print("My hand:")
print(my_hand)
```

Como es de esperar, al imprimir `your_hand` obtenemos `Ac 2c` y al imprimir `my_hand` vemos `3c 4c 5c`.

Finalmente llamamos al método `clear()` de `my_hand` e imprimimos `my_hand` una última vez:

```
my_hand.clear()
print("\nMy hand after clearing it:")
print(my_hand)
```

Por supuesto, por pantalla deberíamos ver la cadena `"<empty>"`.

9.10. HERENCIA.

Uno de los elementos clave en POO es el concepto de **herencia**, el cual nos permite construir una nueva clase basándonos en una clase ya existente. Al hacerlo, la nueva clase automáticamente obtiene (o hereda) todos los métodos y atributos de la clase existente en la que se ha basado.

AMPLIAR UNA CLASE MEDIANTE HERENCIA.

La herencia es especialmente útil cuando queremos crear una versión más especializada de una clase ya existente. Como acabamos de explicar, al crear una nueva clase basándonos en una clase ya existente, la

nueva clase obtiene automáticamente todos los métodos y atributos de la clase previa. Pero también podemos agregar métodos y atributos adicionales a la clase nueva para extender la funcionalidad de los objetos que instanciemos a partir de ella.

Por ejemplo, imaginemos que nuestra clase `Race_car` define un coche de carreras con unos métodos `stop()` y `go()`. Podríamos crear una nueva clase para un coche de carreras especializado que sea capaz de desplegar un alerón aerodinámico, basándonos en la clase preexistente `Race_car`. Nuestra nueva clase heredará automáticamente los métodos `stop()` y `go()` de `Race_car`, y por consiguiente, todo lo que tendremos que hacer será añadir el nuevo método para desplegar el alerón, y nuestra nueva clase ya estará acabada.

A modo de ejemplo práctico, vamos a escribir una segunda versión para el programa del juego de cartas que desarrollamos en la sección previa. Esta nueva versión introduce la clase `Deck`, para describir un mazo de cartas. Sin embargo, y al contrario de cualquier otra clase que hayamos visto hasta ahora, no vamos a escribir la clase `Deck` desde cero, sino que la basaremos en una clase preexistente: la clase `Hand`. Como resultado, `Deck` heredará automáticamente todos los métodos de `Hand`. Vamos a crear `Deck` de esta forma, porque un mazo de cartas es esencialmente una mano de cartas especializada. En efecto, es una mano, pero con algunos comportamientos extra. Un mazo puede hacer lo mismo que una mano: Es una colección de cartas, le puede dar una carta a otra mano, etc. Pero además, un mazo pude hacer unas cuantas cosas que una mano no puede: Un mazo puede barajarse, y puede repartir cartas a varias manos. Nuestra segunda versión del juego de cartas creará un mazo que repartirá cartas a dos manos diferentes. El código para esta segunda versión del juego de cartas se muestra a continuación. Cópialo en una nueva ventana del editor de archivos, y guarda el programa como `playing_cards 2.py`:

```
class Card():
    """ A playing card. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7",
             "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]

    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

    def __str__(self):
        rep = self.rank + self.suit
        return rep

class Hand():
    """ A hand of playing cards. """
    def __init__(self):
        self.cards = []

    def __str__(self):
        if self.cards:
            rep = ""
            for card in self.cards:
                rep += str(card) + "\t"
        else:
            rep = "<empty>"
        return rep

    def clear(self):
        self.cards = []
```

```
    def add(self, card):
        self.cards.append(card)

    def give(self, card, other_hand):
        self.cards.remove(card)
        other_hand.add(card)

class Deck(Hand):
    """ A deck of playing cards. """
    def populate(self):
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                self.add(Card(rank, suit))

    def shuffle(self):
        import random
        random.shuffle(self.cards)

    def deal(self, hands, per_hand = 1):
        for rounds in range(per_hand):
            for hand in hands:
                if self.cards:
                    top_card = self.cards[0]
                    self.give(top_card, hand)
                else:
                    print("Can't continue deal. Out of cards!")


# main
deck1 = Deck()
print("Created a new deck.")
print("Deck:")
print(deck1)

deck1.populate()
print("\nPopulated the deck.")
print("Deck:")
print(deck1)

deck1.shuffle()
print("\nShuffled the deck.")
print("Deck:")
print(deck1)

my_hand = Hand()
your_hand = Hand()
hands = [my_hand, your_hand]
deck1.deal(hands, per_hand = 5)
print("\nDealt 5 cards to my hand and your hand.")
print("My hand:")
print(my_hand)
print("Your hand:")
print(your_hand)
print("Deck:")
print(deck1)

deck1.clear()
print("\nCleared the deck.")
print("Deck:", deck1)
```

Crear una clase de base.

Comenzamos el programa como la versión antigua: Las dos primeras clases, `Card` y `Hand`, son las mismas que antes. En particular, la clase `Hand` será la base para crear la nueva clase `Deck`.

Heredar de una clase base.

Lo siguiente que hacemos es crear la clase `Deck`. Como podemos ver en la cabecera de la clase, `Deck` está basada en la clase `Hand`:

```
class Deck(Hand):
```

Se dice que `Hand` es una **clase base** porque `Deck` se basa en ella. También se dice que `Deck` es una **clase derivada**, porque parte de su definicicón proviene de `Hand`. Como resultado de esta relación, `Deck` hereda todos los métodos de `Hand`. Por consiguiente, incluso aunque no definiésemos ni un solo método nuevo para esta clase, los objetos `Deck` todavía tendrían disponibles todos los métodos definidos en `Hand`, a saber:

- `__init__()`
- `__str__()`
- `clear()`
- `add()`
- `give()`

Si sirve de ayuda, podemos imaginar que, como consecuencia de la herencia, hemos copiado y pegado todos los métodos de `Hand` directamente en la defincición de `Deck`.

Ampliar una clase derivada.

Ahora podemos ampliar la funcionalidad de la clase derivada definiendo métodos adicionales para ella, y eso es exactamente lo que hemos hecho en la defincición de la clase `Deck`. Esto es, además de los métodos heredados de la clase `Hand`, la clase `Deck` también dispone de los siguientes nuevos métodos:

- `populate()`
- `shuffle()`
- `deal()`

El método `populate()` crea una baraja completa con todas las cartas posibles, el método `shuffle()` baraja las cartas para desordenarlas, y el método `deal()` reparte tantas cartas como indiquemos, entre tantas manos como especifiquemos.

En lo que al código externo a esta clase concierne, cualquier método de `Deck` es tan válido como los demás, independientemente de que sea un método heredado de `Hand`, o un método nuevo. Y todos los métodos de `Deck` se invocan de la misma forma, esto es, con la notación basada en puntos.

Usar una clase derivada.

Lo primero que hacemos en el programa principal es instanciar un nuevo objeto `Deck`:

```
# main
deck1 = Deck()
```

Si echamos un vistazo a la clase, parece que no hay un método constructor en `Deck`. Pero esto no es cierto, porque `Deck` hereda en constructor de `Hand`, y es ese método al que se llama directamente nada más crear el objeto `Deck`. Como resultado, el nuevo objeto `Deck` recibe un atributo `cards` que se inicializa a una

lista vacía, como cualquier otro objeto `Hand` recién creado. Finalmente, la sentencia de asignación asigna el nuevo objeto a la variable `deck1`.

Una vez disponemos de un nuevo mazo (pero vacío), lo imprimimos:

```
print("Created a new deck.")
print("Deck:")
print(deck1)
```

Aunque no hemos definido explícitamente un método `__str__()` para `Deck`, de nuevo `Deck` hereda este método de `Hand`. Como el mazo está vacío, este código muestra el texto `<empty>`. Hasta ahora, nuestro mazo se parece mucho a una mano. Esto es así porque un mazo un tipo especializado de mano. Recordemos que, debido a la herencia, un mazo puede hacer lo mismo que una mano, más algunas cosas adicionales.

Pero un mazo vacío es un poco inútil, así que ahora llamamos al método `populate()`, que rellena el mazo con las 52 cartas de una baraja americana tradicional:

```
deck1.populate()
```

En esta ocasión, el mazo por fin ha hecho algo que una mano no podía. Esto es así porque el método `populate()` es un nuevo método que hemos definido para la clase `Deck`. El método `populate()` itera a través de las 52 combinaciones posibles entre los valores de `Card.SUITS` y `Card.RANKS`. Para cada combinación, el método crea un nuevo objeto `Card` y lo añade al mazo de cartas.

A continuación, imprimimos el mazo de cartas:

```
print("\nPopulated the deck.")
print("Deck:")
print(deck1)
```

Esta vez se muestran por pantalla las 52 cartas del mazo. Pero si nos fijamos con atención, están ordenadas. Para hacer las cosas más interesantes, vamos a barajar el mazo:

```
deck1.shuffle()
```

El método `suffle()` de `Deck` importa el módulo `random`, y entonces llama a la función `random.shuffle()`, pasándole como argumento el atributo `cards`. Como imaginaremos, la función `random.shuffle()` sirve para cambiar el orden de los elementos de una lista de forma aleatoria. Así pues, los elementos de la lista `cards` resultan desordenados, tal y como queríamos.

Ahora que las cartas están en un orden aleatorio, mostramos nuevamente el mazo:

```
print("\nShuffled the deck.")
print("Deck:")
print(deck1)
```

Después, creamos dos objetos `Hand` y los ponemos en una lista, que asignamos a la variable `hands`:

```
my_hand = Hand()
your_hand = Hand()
hands = [my_hand, your_hand]
```

Ahora repartimos a cada mano cinco cartas:

```
deck1.deal(hands, per_hand = 5)
```

El método `deal()` es un nuevo método de `Deck`. Este método recibe dos argumentos: Una lista de manos y el número de cartas a repartir a cada mano. El método le da una carta del mazo a cada mano. Si el mazo se ha quedado sin cartas, el método imprime el mensaje `Can't continue deal. Out of cards!`. El método repite este proceso tantas veces como número de cartas deben repartirse a cada mano. Así pues, la línea de código previa reparte cinco cartas a cada mano (`my_hand` y `your_hand`).

Para ver el resultado del reparto, imprimo cada mano y el mazo una vez más:

```
print("\nDealt 5 cards to my hand and your hand.")
print("My hand:")
print(my_hand)
print("Your hand:")
print(your_hand)
print("Deck:")
print(deck1)
```

Analizando la salida, podemos ver que cada mano tiene `5` cartas, y que el mazo solo tiene `42`.

Finalmente, ponemos el mazo en su estado inicial limpiándolo:

```
deck1.clear()
print("\nCleared the deck.")
```

, y terminamos imprimiendo el mazo una última vez:

```
print("Deck:", deck1)
```

ALTERAR EL COMPORTAMIENTO DE LOS MÉTODOS HEREDADOS.

Acabamos de ver cómo extender la funcionalidad de una clase añadiendo nuevos métodos a una clase derivada. Pero también podemos redefinir la forma en la que un método heredado de una clase base funciona en una clase derivada. A esta técnica se la llama **sobrecargar** el método. Para sobrecargar un método de una clase base, tenemos dos opciones: Podemos crear un método con una funcionalidad completamente nueva, o podemos incorporar nuevas funcionalidades el método de la clase base que estamos sobrecargando.

Tomemos como ejemplo nuestra clase `Race_car`. Digamos que su método `stop()` simplemente aplica los frenos al coche. Si queremos crear una nueva clase de coche de carreras que frene más rápido (por ejemplo, desplegando un paracaídas detrás del coche), podríamos derivar una nueva clase `Parachute_race_car` a partir de la clase base `Race_car`, y sobrecargar su método `stop()`. De esta forma, podríamos escribir el nuevo método `stop()` de forma que llamase al método `stop()` de la clase `Race_car` original (que aplica los frenos al coche), y añadirle la acción de desplagar el paracaídas del coche.

A modo de ejemplo práctico vamos a escribir una tercera versión para el programa del juego de cartas que hemos venido desarrollando en secciones previas. Esta versión deriva dos nuevas clases de cartas a partir de la clase `Card` con la que hemos estado trabajando hasta ahora. La primera clase nueva define las cartas que no se pueden imprimir por pantalla. De forma más precisa, cuando imprimamos un objeto de esta clase, se mostrará el texto `<unprintable>`. La segunda clase nueva define las cartas que pueden estar o bien bocarriba o bien bocabajo. Cuando imprimamos un objeto de esta clase, habrá dos resultados posibles: Si la carta está bocarriba, se imprime exactamente igual que cualquier objeto de la clase `Card`. Pero si está bocabajo, se muestra el texto `XX`.

Escribe este código en el editor de archivos y guárdalo como `playing_cards 3.py`:

```
class Card():
    """ A playing card. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7",
             "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]

    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

    def __str__(self):
        rep = self.rank + self.suit
        return rep


class Unprintable_Card(Card):
    """ A Card that won't reveal its rank or suit when printed. """
    def __str__(self):
        return "<unprintable>"


class Positionable_Card(Card):
    """ A Card that can be face up or face down. """
    def __init__(self, rank, suit, face_up = True):
        super(Positionable_Card, self).__init__(rank, suit)
        self.is_face_up = face_up

    def __str__(self):
        if self.is_face_up:
            rep = super(Positionable_Card, self).__str__()
        else:
            rep = "XX"
        return rep

    def flip(self):
        self.is_face_up = not self.is_face_up


#main
card1 = Card("A", "c")
card2 = Unprintable_Card("A", "d")
card3 = Positionable_Card("A", "h")

print("Printing a Card object:")
print(card1)

print("\nPrinting an Unprintable_Card object:")
print(card2)

print("\nPrinting a Positionable_Card object:")
print(card3)
print("Flipping the Positionable_Card object.")
card3.flip()
print("Printing the Positionable_Card object:")
print(card3)
```

Crear una clase base.

Para derivar una nueva clase, necesitamos comenzar con una clase base. Para este programa, hemos usado como clase base la clase `Card` con la que hemos trabajado en las dos versiones previas.

Sobrecargar los métodos de la clase base.

A continuación derivamos una nueva clase para las cartas que no se pueden imprimir. El encabezado de la clase es el habitual al trabajar con herencia:

```
class Unprintable_Card(Card):
```

De este encabezado, sabemos que la clase derivada `Unprintable_Card` hereda todos los métodos de `Card`. Pero ahora podemos cambiar el comportamiento de un método heredado volviéndolo a definir en la clase derivada. Y eso es exactamente lo que hemos hecho en el resto de la definición del método:

```
class Unprintable_Card(Card):
    """ A Card that won't reveal its rank or suit when printed. """
    def __str__(self):
        return "<unprintable>"
```

La clase `Unprintable_Card` hereda el método `__str__()` de su clase base `Card`. Pero aquí hemos definido un nuevo método `__str__()` para `Unprintable_Card` que sobrecarga (o reemplaza) el método heredado. Siempre que creamos un método en una clase derivada con el mismo nombre que un método heredado, estamos sobrecargando el método heredado en la nueva clase. Así pues, cuando imprimamos un objeto `Unprintable_Card`, se mostrará el texto `<unprintable>`.

Lo que hagamos en la clase derivada no tiene efecto en la clase base. A la clase base no le importa si hemos derivado una nueva clase a partir de ella, o si hemos sobrecargado un método heredado en la clase nueva. Esto significa que cuando imprimamos un objeto `Card`, se mostrará por pantalla como siempre lo hace.

Llamar a los métodos de la clase base.

En ocasiones, aún cuando sobrecargamos el método de la clase base, necesitamos incorporar la funcionalidad del método heredado. Por ejemplo, imaginar que queremos crear un nuevo tipo de carta basada en la clase `Card`. Suponer que necesitamos que los objetos de esta nueva clase tengan un atributo que indique si esta carta está bocarriba. Para ello, debemos sobrecargar el método constructor heradado de `Card` con un nuevo constructor que cree un atributo `face_up` (bocarriba). Sin embargo, también querremos que nuestro nuevo constructor cree y fije los valores de los atributos `rank` y `suit`, de la misma forma que lo hace el constructor de la clase base `Card`. En vez de reescribir el código del constructor de `Card`, podríamos simplemente invocarlo desde dentro de mi nuevo constructor. De esta forma, sería el constructor de `Card` el que crearía e inicializaría los atributos `rank` y `suit` de los objetos de la nueva clase. De vuelta al método constructor de la nueva clase, podríamos añadir el atributo `face_up` que indica si la carta está o no bocarriba. Bueno, ése es exactamente el enfoque que hemos aplicado en la clase `Positionable_Card`:

```
class Positionable_Card(Card):
    """ A Card that can be face up or face down. """
    def __init__(self, rank, suit, face_up = True):
        super(Positionable_Card, self).__init__(rank, suit)
        self.is_face_up = face_up
```

La función `super()` que hemos usado dentro del nuevo método constructor nos permite llamar al método constructor de la clase base (también llamada **superclase**, de ahí el nombre de la función). La línea

`super(Positionable_Card, self).__init__(rank, suit)` llama al método `__init__()` de `Card` (la superclase de `Positionable_Card`). El primer argumento en la llamada a la función `super()`, que es `Positionable_Card`, dice que queremos llamar al método de la superclase (o clase base) de `Positionable_Card`, que es `Card`. El siguiente argumento, `self`, le pasa una referencia al objeto `Positionable_Card` recién instanciado, para que el código en `Card` pueda obtener el objeto al que añadirle los atributos `rank` y `suit`. La siguiente parte de la sentencia,`__init__(rank, suit`, le dice a Python que queremos llamar al método constructor de `Card` y que queremos pasarle los valores de `rank` y `suit`.

El siguiente método de `Positionable_Card` también sobrecarga un método heredado de `Card` y llama al método sobrecargado:

```
def __str__(self):
    if self.is_face_up:
        rep = super(Positionable_Card, self).__str__()
    else:
        rep = "XX"
    return rep
```

En primer lugar, el método `__str__()` comprueba si el atributo `face_up` del objeto es `True` (lo que significa que la carta está bocarriba). Si es así, la representación tipo cadena de la carta es igual a la cadena devuelta por el método `__str__()` de `Card`, al que llamamos con la función `super()` pasándole como argumento el objeto `Positionable_Card`. En otras palabras, si la carta está bocarriba, la carta se imprime como cualquier otro objeto de la clase Card. Sin embargo, si la carta no está bocarriba, la representación en forma de cadena que se devuelve es `"XX"`.

El último método de la clase no sobrecarga un método heredado. Simplemente extiende la definición de esta nueva clase:

```
def flip(self):
    self.is_face_up = not self.is_face_up
```

El método le da la vuelta a una carta cambiando el valor de su atributo `face_up`. Si el atributo `face_up` de un objeto está a `True`, la llamada al método `flip()` fija su valor a `False`. Y si el atributo `face_up` de un objeto está a `False`, la llamada a `flip()` lo cambia a `True`.

Usar una clase derivada.

En el programa principal creamos tres objetos: un objeto `Card`, otro objeto `Unprintable_Card`, y un objeto `Positionable_Card`:

```
#main
card1 = Card("A", "c")
card2 = Unprintable_Card("A", "d")
card3 = Positionable_Card("A", "h")
```

A continuación, imprimimos el objeto `Card`:

```
print("Printing a Card object:")
print(card1)
```

Esto funciona exactamente igual que en los programas previos, por lo que se imprime el texto `Ac`.

Lo siguiente que hacemos es imprimir el objeto `Unprintable_Card`:

```
print("\nPrinting an Unprintable_Card object:")
print(card2)
```

Incluso aunque el objeto tiene el atributo `rank` fijado a `"A"` y al atributo `suit` fijado a `"d"`, la impresión del objeto muestra `<unprintable>` por pantalla, porque la clase `Unprintable_Card` sobrecarga su método `__str__()` heredado con uno que siempre retorna la cadena `"<unprintable>"`.

Las siguientes dos líneas imprimen un objeto `Positionable_Card`:

```
print("\nPrinting a Positionable_Card object:")
print(card3)
```

Como el atributo `face_up` del objeto está a `True`, el método `__str__()` del objeto llama al método `__str__()` de `Card`, y se imprime el texto `Ah` por pantalla.

Después, llamamos al método `flip()` de `Positionable_Card`:

```
print("Flipping the Positionable_Card object.")
card3.flip()
```

Como resultado, el atributo `face_up` del objeto se fija a `False`.

Las dos últimas líneas imprimen nuevamente el objeto `Positionable_Card`:

```
print("Printing the Positionable_Card object:")
print(card3)
```

En esta ocasión, se imprime el texto `XX`, porque el atributo `face_up` del objeto está a `False`.

9.11. CREAR E IMPORTAR MÓDULOS.

Puede que nos sorprenda ver una sección sobre módulos a estas alturas, habida cuenta de que llevamos importando y usando módulos casi desde el principio (por ejemplo, los módulos `math`, `random`, etc.). Todos estos módulos pertenecen a la llamada **librería estándar de Python**, un conjunto de módulos incluidos en toda instalación de Python.

Sin embargo, un aspecto muy importante de la programación con Python es que nosotros también podemos crear, usar, e incluso compartir nuestros propios módulos. Crear nuestros propios módulos aporta importantes beneficios:

En primer lugar, al crear nuestros módulos podemos reusar código, lo que puede ahorranos un montón de tiempo y esfuerzo. Por ejemplo, podríamos reusar las clases `Card`, `Hand`, y `Deck` que hemos escrito en secciones previas para crear muchos tipos de juegos de cartas sin tener que reinventar la carta básica, la mano, y el mazo cada una de las veces.

En segundo lugar, al dividir un programa en módulos lógicos, los programas largos son más fáciles de gestionar. Hasta ahora, todos los programas con los que hemos trabajado estaban contenidos en un solo archivo. Como eran relativamente cortos, esto no era un problema. Pero imaginemos un programa con miles (o incluso decenas de miles) de líneas de código. Trabajar con un programa de este tamaño en un único y gigantesco archivo sería una pesadilla. (Por cierto, los proyectos profesionales suelen tener estas dimensiones). Dividir el código en módulos facilita a los distintos miembros de un equipo de desarrollo de software repartirse y abordar las distintas tareas del proyecto.

En tercer lugar, la creación de modulos nos permitirá compartir nuestros programas. Si creamos un módulo de gran utilidad, podemos enviárselo por correo a un amigo, quien podrá usarlo de la misma forma que usa un módulo de la biblioteca estándar de Python.

Para aprender a crear módulos, vamos a escribir un programa para un juego muy sencillo: El programa pregunta cuántos jugadores van a participar en el juego, y a continuación obtiene el nombre de cada jugador. Finalmente, el programa asigna a cada jugador una puntuación aleatoria, y muestra el resultado. No es un juego demasiado interesante, pero el hecho importante es que usará un nuevo módulo que contiene funciones y una clase que nosotros habremos creado.

ESCRIBIR MÓDULOS.

Normalmente, a continuación mostraríamos el código del programa del juego sencillo, para copiarlo y analizarlo. Pero en esta sección primero vamos a ver el código del módulo que usa este juego.

Para escribir un módulo hacemos lo mismo que al escribir cualquier otro programa de Python. Sin embargo, al escribir un módulo deberíamos construir una colección de componentes de programación (como funciones y clases) que estén relacionados, y guardarlos en un solo archivo que después importaremos en nuestro programa.

Aquí hemos creado un módulo básico, llamado `games`, que contiene dos funciones y una clase que podrían ser útiles para crear casi cualquier juego. Copia el código del módulo en el editor de archivos y guárdalo como `games.py`:

```
class Player():
    """ A player for a game. """
    def __init__(self, name, score = 0):
        self.name = name
        self.score = score

    def __str__(self):
        rep = self.name + ":\t" + str(self.score)
        return rep

def ask_yes_no(question):
    """Ask a yes or no question."""
    response = None
    while response not in ("y", "n"):
        response = input(question)
    return response

def ask_number(question, low, high):
    """Ask for a number within a range."""
    response = None
    while response not in range(low, high):
        response = int(input(question))
    return response


if __name__ == "__main__":
    print("You ran this module directly (and did not 'import' it).")
    input("\n\nPress the enter key to exit.")
```

Este módulo se llama `games` porque lo hemos guardado con el nombre `games.py`. Los módulos se nombran (y se importan) basándose en sus nombres de archivo.

El funcionamiento de este módulo es sencillo: La clase `Player` define un objeto con dos atributos, `name` y `score`, cuyos valores se establecen en el método constructor. El otro método de la clase, `__str()__`, simplemente devuelve una representación tipo cadena del objeto para poder imprimirlo por pantalla.

La función `ask_yes_no()` sive para hacer una pregunta cuya única respuesta posible es `"y"` (sí) o `"n"` (no), mientras que la función `ask_number()` sirve para hacer una pregunta cuya única respuesta posible es un número dentro de un rango de valores posibles.

La última parte del programa introduce una nueva idea relacionada con los módulos. La condición de la sentencia `if`, a saber, `__name__ == "__main__"`, es cierta si el programa se ejecuta directamente, y es falsa si el archivo se importa como un módulo. Así pues, si `games.py` se ejecuta directamente, se muestra un mensaje diciéndole al usuario que este archivo está pensado para ser importado, y no para ejecutarse directamente.

IMPORTAR MÓDULOS.

Una vez analizado el módulo `games`, vamos a presentar el código del programa del juego sencillo. Escribe lo siguiente en el editor de archivos, y guárdalo como `simple_game.py`:

```
import games, random

print("Welcome to the world's simplest game!\n")

again = None
while again != "n":
    players = []
    num = games.ask_number(question = "How many players? (2 - 5): ",
                           low = 2, high = 5)
    for i in range(num):
        name = input("Player name: ")
        score = random.randrange(100) + 1
        player = games.Player(name, score)
        players.append(player)

    print("\nHere are the game results:")
    for player in players:
        print(player)

    again = games.ask_yes_no("\nDo you want to play again? (y/n): ")
```

Para importar un módulo propio, usamos la sentencia `import` exactamente igual que para importar un módulo estándar de Python. De hecho, en este programa hemos importado el módulo `games` junto con el familiar módulo `random` con una sola sentencia import:

```
import games, random
```

> NOTA: Si un módulo propio no está en la misma carpeta que el programa que lo importa, Python será incapaz de encontralo. Por supuesto hay formas de evitar este problema. Incluso es posible instalar módulos propios de forma que estén disponibles de forma global para un cierto sistema, de la misma forma que lo están los módulos integrados de Python; sin embargo, esto requeriría un procedimiento de instalación especial que está más allá de nuestros actuales propósitos. Así que, por ahora, debemos asegurarnos de que cualquier módulo propio que vayamos a importar esté situado en la misma carpeta que el programa que lo importa.

USAR FUNCIONES Y CLASES IMPORTADAS.

En el resto del programa básicamente nos hemos dedicado a usar los módulos que hemos importado. Después de dar la bienvenida a los jugadores y configurar un bucle `while`, preguntamos cuántos jugadores participarán en el juego.

Para obtener el número de jugadores llamamos a la función `ask_number()` del módulo `games`. De la misma forma que llamamos a las funciones de los módulos integrados de Python, para llamar a una función de un módulo propio usamos la notación basada en puntos, especificando el nombre del módulo, seguido de un punto, seguido del nombre de la función.

A continuación, y para cada jugador, obtenemos el nombre del jugador, y generamos una puntuación aleatoria entre 1 y 100 llamando a la función `randrange()` del módulo `random`. Después, creamos un objeto `Player` usando ese nombre y esa puntuación. Como la clase `Player` está definida en el módulo `games`, debemos usar la notación basada en puntos e incluir el nombre del módulo antes del nombre de la clase. Luego, añadimos a este nuevo objeto `Player` a una lista de jugadores.

Ahora imprimimos cada jugador creado.

Para terminar preguntamos si los jugadores quieren jugar otra partida. Usamos la función `ask_yes_no()` del módulo `games` para obtener la respuesta.

9.12. PROYECTO: BLACK JACK.

El proyecto final de este capítulo es una versión simplificada del juego de cartas blackjack. El juego funcionará de la siguiente manera: A los jugadores se les reparten unas cartas con ciertos puntos. Cada jugador tratará de llegar a 21 puntos, sin pasarse. Las cartas numeradas valen un número de puntos igual al valor de la carta. Los ases cuentan 1 u 11 puntos (lo que sea mejor para el jugador), y cualquier sota, reina, o rey cuenta 10 puntos.

El ordenador será el que reparta, y competirá contra un máximo de 7 jugadores. Al principio de la ronda, el ordenador reparte dos cartas a todos los participantes (él mismo incluido). Los jugadores pueden ver todas sus cartas, y el ordenador incluso muestra el total de puntos. Sin embargo, una de las cartas del ordenador empieza oculta a los jugadores.

A continuación, cada jugador tiene la opción de recibir cartas adicionales. Todo jugador puede tomar una carta más cada vez, siempre que así lo desee. Pero si la puntuación total de un jugador excede de 21 (a lo que en la jerga se le llama "burst"), ese jugador pierde. Si todos los jugadores pierden, el ordenador revela su primera carta y la ronda termina. En caso contrario, la partida continúa. El ordenador debe tomar cartas adicionales siempre que su puntuación total sea menor que 17. Si el ordenador se pasa, todos los jugadores que no se hayan pasado ganan. En caso contrario, la puntuación total de cada jugador que no se haya pasado se compara con la puntuación del ordenador. Si la puntuación total del jugador es más alta, el jugador gana, y si es menor, pierde. Si los totales son iguales, el jugador empata con el ordenador (a lo que se le denomina "push").

Llegados a este punto, ya sabemos crear clases de Python para representar cartas en juego, manos, y mazos. Vamos a ver cómo combinar todas estas clases en un programa que cree un juego de cartas de verdad.

EL MÓDULO CARDS.

Para escribir el programa del blackjack, vamos a crear un módulo `cards` basado en los programas de cartas de las secciones previas. Las clases `Hand` y `Deck` son exactamente iguales que las del programa `playing_cards 2.py`. La nueva clase `Card` representa la misma funcionalidad que la clase `Positionable_Card` del programa `playing_cards 3.py`. Copia el código del módulo en el editor de archivos, y guárdalo con el nombre `cards.py`:

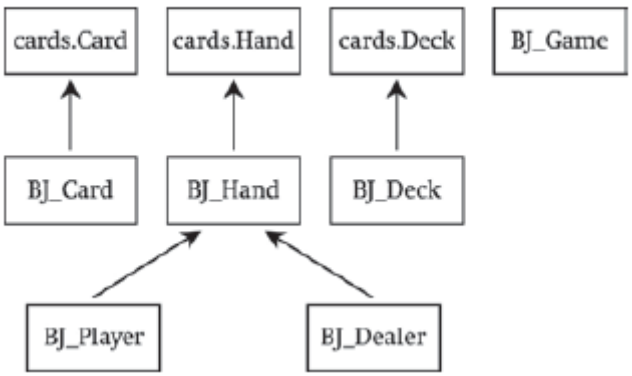
```
class Card():
    """ A playing card. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7",
             "8", "9", "10", "J", "Q", "K"]
    SUITS = ["c", "d", "h", "s"]

    def __init__(self, rank, suit, face_up = True):
        self.rank = rank
        self.suit = suit
        self.is_face_up = face_up

    def __str__(self):
        if self.is_face_up:
            rep = self.rank + self.suit
        else:
            rep = "XX"
        return rep

    def flip(self):
        self.is_face_up = not self.is_face_up


class Hand():
    """ A hand of playing cards. """
    def __init__(self):
        self.cards = []

    def __str__(self):
        if self.cards:
           rep = ""
           for card in self.cards:
               rep += str(card) + "\t"
        else:
            rep = "<empty>"
        return rep

    def clear(self):
        self.cards = []

    def add(self, card):
        self.cards.append(card)

    def give(self, card, other_hand):
        self.cards.remove(card)
        other_hand.add(card)
```

```
class Deck(Hand):
    """ A deck of playing cards. """
    def populate(self):
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                self.add(Card(rank, suit))

    def shuffle(self):
        import random
        random.shuffle(self.cards)

    def deal(self, hands, per_hand = 1):
        for rounds in range(per_hand):
            for hand in hands:
                if self.cards:
                    top_card = self.cards[0]
                    self.give(top_card, hand)
                else:
                    print("Can't continue deal. Out of cards!")



if __name__ == "__main__":
    print("This is a module with classes for playing cards.")
    input("\n\nPress the enter key to exit.")
```

DISEÑAR LAS CLASES.

Antes de empezar a escribir un proyecto con varias clases, suele ser útil hacer una lista de las clases que vamos a necesitar, e incluir una breve descripción de ellas. Además de describir las clases con palabras, también vale la pena dibujar un diagrama para visualizar las relaciones de herencia entre ellas.

Class	Base Class	Description
BJ_Card	cards.Card	A blackjack playing card. Define an attribute `value` to represent the point value of a card.
BJ_Deck	cards.Deck	A blackjack deck. A collection of `BJ_Card` objects.
BJ_Hand	cards.Hand	A blackjack hand. Define an attribute `total` to represent the point total of a hand. Define an attribute `name` to represent the owner of the hand.
BJ_Player	BJ_Hand	A blackjack player.
BJ_Dealer	BJ_Hand	A blackjack dealer.
BJ_Game	object	A blackjack game. Define an attribute `deck` for a `BJ_Deck` object. Define an attribute `dealer` for a `BJ_Dealer` object. Define an attribute `players` for a list of `BJ_Player` objects.



ESCRIBIR EL PSEUDOCÓDIGO DE UNA RONDA DEL JUEGO.

Lo siguiente que podemos hacer para planificar nuestra tarea es escribir con palabras cómo programaremos una ronda del juego. Esto puede ser útil para ver cómo interaccionan los distintos objetos. Un ejemplo de pseudocódigo para una ronda de blackjack sería el siguiente:

```
Repartir al ordenador y a cada jugador 2 cartas iniciales

Para cada jugador
     Mientras el jugador pida otra carta y no se pase
          Repartir otra carta adicional al jugador

Si no quedan más jugadores
     Mostrar las dos cartas del ordenador

En caso contrario
     Mientras el ordenador pueda pedir más catas y no se pase
          Repartir otra carta adicional al ordenador

     Si el ordenador se pasa
          Para cada jugador que todavía esté jugando
               El jugador gana

     En caso contrario
          Para cada jugador que todavía esté jugando
               Si el total del jugador es mayor que el total del ordenador
                    El jugador gana
               Si el total del jugador es menor que el total del ordenador
                    El jugadro pierde
               En otro caso
                    El jugador empata
```

EL PROGRAMA DEL BLACKJACK.

Ahora que tenemos una visión global de la estructura del programa, vamos con el código. Escribe lo siguiente en el editor de archivos, y guárdalo como `blackjack.py`:

```
import cards, games

class BJ_Card(cards.Card):
    """ A Blackjack Card. """
    ACE_VALUE = 1

    @property
    def value(self):
        if self.is_face_up:
            v = BJ_Card.RANKS.index(self.rank) + 1
            if v > 10:
                v = 10
        else:
            v = None
        return v

class BJ_Deck(cards.Deck):
    """ A Blackjack Deck. """
    def populate(self):
        for suit in BJ_Card.SUITS:
            for rank in BJ_Card.RANKS:
                self.cards.append(BJ_Card(rank, suit))
```

```
class BJ_Hand(cards.Hand):
    """ A Blackjack Hand. """
    def __init__(self, name):
        super(BJ_Hand, self).__init__()
        self.name = name

    def __str__(self):
        rep = self.name + ":\t" + super(BJ_Hand, self).__str__()
        if self.total:
            rep += "(" + str(self.total) + ")"
        return rep

    @property
    def total(self):
        # if a card in the hand has value of None, then total is None
        for card in self.cards:
            if not card.value:
                return None

        # add up card values, treat each Ace as 1
        t = 0
        for card in self.cards:
            t += card.value

        # determine if hand contains an Ace
        contains_ace = False
        for card in self.cards:
            if card.value == BJ_Card.ACE_VALUE:
                contains_ace = True

        # if hand contains Ace and total is low enough, treat Ace as 11
        if contains_ace and t <= 11:
            # add only 10 since we've already added 1 for the Ace
            t += 10

        return t

    def is_busted(self):
        return self.total > 21


class BJ_Player(BJ_Hand):
    """ A Blackjack Player. """
    def is_hitting(self):
        response = games.ask_yes_no("\n" + self.name +
                                    ", do you want a hit? (Y/N): ")
        return response == "y"

    def bust(self):
        print(self.name, "busts.")
        self.lose()

    def lose(self):
        print(self.name, "loses.")

    def win(self):
        print(self.name, "wins.")

    def push(self):
        print(self.name, "pushes.")
```

```
class BJ_Dealer(BJ_Hand):
    """ A Blackjack Dealer. """
    def is_hitting(self):
        return self.total < 17

    def bust(self):
        print(self.name, "busts.")

    def flip_first_card(self):
        first_card = self.cards[0]
        first_card.flip()


class BJ_Game():
    """ A Blackjack Game. """
    def __init__(self, names):
        self.players = []
        for name in names:
            player = BJ_Player(name)
            self.players.append(player)

        self.dealer = BJ_Dealer("Dealer")

        self.deck = BJ_Deck()
        self.deck.populate()
        self.deck.shuffle()

    @property
    def still_playing(self):
        sp = []
        for player in self.players:
            if not player.is_busted():
                sp.append(player)
        return sp

    def __additional_cards(self, player):
        while not player.is_busted() and player.is_hitting():
            self.deck.deal([player])
            print(player)
            if player.is_busted():
                player.bust()

    def play(self):
        # deal initial 2 cards to everyone
        self.deck.deal(self.players + [self.dealer], per_hand = 2)
        self.dealer.flip_first_card()    # hide dealer's first card
        for player in self.players:
            print(player)
        print(self.dealer)

        # deal additional cards to players
        for player in self.players:
            self.__additional_cards(player)

        self.dealer.flip_first_card()    # reveal dealer's first
```

```
        if not self.still_playing:
            # since all players have busted, just show the dealer's hand
            print(self.dealer)
        else:
            # deal additional cards to dealer
            print(self.dealer)
            self.__additional_cards(self.dealer)

            if self.dealer.is_busted():
                # everyone still playing wins
                for player in self.still_playing:
                    player.win()
            else:
                # compare each player still playing to dealer
                for player in self.still_playing:
                    if player.total > self.dealer.total:
                        player.win()
                    elif player.total < self.dealer.total:
                        player.lose()
                    else:
                        player.push()

        # remove everyone's cards
        for player in self.players:
            player.clear()
        self.dealer.clear()


def main():
    print("\t\tWelcome to Blackjack!\n")

    names = []
    number = games.ask_number("How many players? (1-7): ", low = 1, high = 8)
    for i in range(number):
        name = input("Enter player name: ")
        names.append(name)
    print()

    game = BJ_Game(names)

    again = None
    while again != "n":
        game.play()
        again = games.ask_yes_no("\nDo you want to play again?: ")


main()
```

El programa es ciertamente largo, así que vamos a ir analizándolo paso a paso.

Importar los módulos cards y games.

En la primera parte del programa, importamos los módulos `cards` y `games`:

```
import cards, games
```

Recordemos que el módulo `games` ya lo creamos en una sección previa del capítulo.

La clase BJ_Card.

La clase `BJ_Card` extiende la definición de lo que es una carta heredando la clase `cards.Card`. En `BJ_Card` hemos creado una nueva *propiedad*, `value`, para el valor en puntos de la carta:

```
class BJ_Card(cards.Card):
    """ A Blackjack Card. """
    ACE_VALUE = 1

    @property
    def value(self):
        if self.is_face_up:
            v = BJ_Card.RANKS.index(self.rank) + 1
            if v > 10:
                v = 10
        else:
            v = None
        return v
```

El método devuelve un número entre 1 y 10, que representa el valor de una carta de blackjack. La primera parte del cálculo se realiza con la expresión `BJ_Card.RANKS.index(self.rank) + 1`. Esta expresión toma el atributo `rank` de un objeto `BJ_Card` (por ejemplo, `"6"`) y halla su índice correspondiente en la lista `BJ_Card.RANKS` mediante el método `index()` (para `"6"` el índice es 5). Finalmente, le suma 1 al resultado, porque los índices de las listas comeinzan contando desde 0. Pero como los valores de `rank` para `"J"`, `"Q"`, y `"K"` son números mayores de 10, cualquier `value` mayor que 10 se fija directamente a 10. Si el atributo `face_up` de un objeto es `False`, todo este proceso se omite y el método devuelve `None`.

Por supuesto, este proceso lo podríamos haber hecho con múltiples sentencias `elif`, pero esta forma es mucho más compacta y elegante.

La clase BJ_Deck.

La clase `BJ_Deck` se usa para crear un mazo de cartas de blackjack. Esta clase es casi idéntica que su clase base, `cards.Deck`. La única diferencia es que hemos sobrecargado el método `populate()` de `cards.Deck` de forma que el nuevo objeto `BJ_Deck` se rellene con objetos `BJ_Card`:

```
class BJ_Deck(cards.Deck):
    """ A Blackjack Deck. """
    def populate(self):
        for suit in BJ_Card.SUITS:
            for rank in BJ_Card.RANKS:
                self.cards.append(BJ_Card(rank, suit))
```

La clase BJ_Hand.

La clase `BJ_Hand`, que está basada en la clase `cards.Hand`, se usa para las manos de blackjack. Hemos sobrecargado el constructor de `cards.Hand` y le hemos añadido un atributo `name` para representar el nombre del jugador que tiene esa mano:

```
class BJ_Hand(cards.Hand):
    """ A Blackjack Hand. """
    def __init__(self, name):
        super(BJ_Hand, self).__init__()
        self.name = name
```

A continuación, sobrecargamos el método heredado `__str__()` para mostrar el valor total de puntos de esa mano:

```
def __str__(self):
    rep = self.name + ":\t" + super(BJ_Hand, self).__str__()
    if self.total:
        rep += "(" + str(self.total) + ")"
    return rep
```

Aquí concatenamos el atributo `name` del objeto con la cadena devuelta por el método `cards.Hand_str__()` del objeto. Después, si la propiedad `total` del objeto no es `None`,[9] concatenamos la representación en forma de cadena con el valor de `total`, y finalmente, devolvemos esa cadena.

A continuación, definimos una propiedad llamada `total`, que representa los puntos totales de una mano de `blackjack`. Si la mano tiene una carta bocabajo dentro de ella, entonces su propiedad `total` es `None`. En caso contrario, el valor se calcula sumando los valores de todas las cartas de la mano:

```
@property
def total(self):
    # if a card in the hand has value of None, then total is None
    for card in self.cards:
        if not card.value:
            return None

    # add up card values, treat each Ace as 1
    t = 0
    for card in self.cards:
          t += card.value

    # determine if hand contains an Ace
    contains_ace = False
    for card in self.cards:
        if card.value == BJ_Card.ACE_VALUE:
            contains_ace = True

    # if hand contains Ace and total is low enough, treat Ace as 11
    if contains_ace and t <= 11:
        # add only 10 since we've already added 1 for the Ace
        t += 10

    return t
```

La primera parte de este método comprueba si alguna de las cartas de la mano de blackjack tiene una propiedad `value` igual a `None` (lo que significaría que la carta está bocabajo). De ser así, el método devuelve `None`. La siguiente parte del método simplemente suma los valores en puntos de todas las cartas de la mano. La última parte del método comprueba si la mano contiene un as. De ser así, determina si el valor en puntos de la carta debería ser 1 u 11.

El último método de `BJ_Hand` es `is_busted()`, que sirve para determinar si la mano se ha pasado. Devuelve `True` si la propiedad `total` del objeto es mayor que 21, y `False` en caso contrario:

```
def is_busted(self):
    return self.total > 21
```

[9] Como veremos más adelante, La propiedad `total` del objeto puede tomar o bien un valor numérico (el total de puntos de una mano), o bien el valor `None` (si una de las cartas de la mano está bocabajo). La condición `if self.total` se evalúa a `False` si su valor es `None`, y a `True` si su valor es numérico (o en general, si su valor es distinto de `None`).

Notar que en este método devolvemos el resultado de la condición `self.total > 21` en vez de asignar el resultado a una variable, para luego devolver la variable. Siempre es posible crear este tipo de sentencias `return` con cualquier condición (y con cualquier expresión, en realidad), y además, es una práctica de programación mucho más elegante. Este tipo de métodos que devuelven `True` o `False` son muy comunes. Como en este caso, a menudo se usan para representar la condición de un objeto con dos posibilidades, como por ejemplo "on" y "off". Estos métodos casi siempre tienen un nombre que comienza con la palabra `is`, como por ejemplo, `is_on()`.

La clase BJ_Player.

La clase `BJ_Player` deriva de `BJ_Hand`, y se usa para representar a los jugadores de blackjack:

```
class BJ_Player(BJ_Hand):
    """ A Blackjack Player. """
    def is_hitting(self):
        response = games.ask_yes_no("\n" + self.name +
                                    ", do you want a hit? (Y/N): ")
        return response == "y"

    def bust(self):
        print(self.name, "busts.")
        self.lose()

    def lose(self):
        print(self.name, "loses.")

    def win(self):
        print(self.name, "wins.")

    def push(self):
        print(self.name, "pushes.")
```

El primer método, `is_hitting()`, devuelve `True` si el jugador quiere otra carta y `False` si no. (En el argot, "hit" significa pedir otra carta). El método `burst()` anuncia que el jugador se ha pasado y llama al método `lose()` del objeto. El método `lose()` anuncia que un jugador pierde. El método `win()` anuncia que un jugador gana. Y el método `push()` anuncia que un jugador y el ordenador han empatado. Los métodos `burst()`, `lose()`, `win()`, y `push()` son tan simples que puede que nos preguntemos por qué existen. Los hemos puesto en una clase porque forman parte de una estructura que permite manejar las situaciones (mucho más complejas) que aparecen cuando a los jugadores se les permite apostar (tarea que se te propone como ejercicio al final del capítulo).

La clase BJ_Dealer.

La clase `BJ_Dealer`, que también está basada en la clase `BJ_Hand`, se usa para el repartidor de cartas de blackjack (papel que desempeña el ordenador):

```
class BJ_Dealer(BJ_Hand):
    """ A Blackjack Dealer. """
    def is_hitting(self):
        return self.total < 17

    def bust(self):
        print(self.name, "busts.")

    def flip_first_card(self):
        first_card = self.cards[0]
        first_card.flip()
```

El primer método, `is_hitting()`, permite decidir si el ordenador recibe o no cartas adicionales. Como el ordenador debe pedir cartas siempre que su total sea menor que 17, el método devuelve `True` si la propiedad `total` del objeto es menor que 17, y `False` en caso contrario. El método `burst()` anuncia que el ordenador se ha pasado. El método `flip_first_card()` le da la vuelta a la primera carta del ordenador.

La clase BJ_Game.

La clase `BJ_Game` se usa para crear un único objeto que representa una partida de blackjack. En su método `play()`, esta clase contiene el código para el desarrollo de una ronda del juego. Sin embargo, la mecánica del juego es lo suficientemente compleja como para que hayamos necesitado crear algunos elementos fuera de este método, incluyendo el método `__additional_cards()`, que se encarga de repartir cartas adicionales a un jugador, y la propiedad `still_playing`, que devuelve una lista de todos los jugadores que áun siguen jugando la ronda actual.

a) El método __init__().

El constructor recibe una lista de nombres y crea un jugador para cada nombre. Este método también crea un repartidor y un mazo de cartas:

```
class BJ_Game():
    """ A Blackjack Game. """
    def __init__(self, names):
        self.players = []
        for name in names:
            player = BJ_Player(name)
            self.players.append(player)

        self.dealer = BJ_Dealer("Dealer")

        self.deck = BJ_Deck()
        self.deck.populate()
        self.deck.shuffle()
```

b) La propiedad still_playing.

Esta propiedad devuelve una lista de todos los jugadores que todavía están jugando (esto es, aquellos que aún no se han pasado en esta ronda):

```
@property
def still_playing(self):
    sp = []
    for player in self.players:
        if not player.is_busted():
            sp.append(player)
    return sp
```

c) El método __additional_cards().

El método `__additional_cards()` reparte cartas adicionales a cualquier jugador o al ordenador. En su parámetro `player`, el método recibe un objeto que puede ser de la clase `BJ_Player` o de la clase `BJ_Dealer`. El método continúa mientras el método `is_bursted()` de ese objeto devuelva `False` y el método `is_hitting()` devuelva `True`. Si el método `is_bursted()` del objeto devuelve `True`, entonces se llama al método `burst()` de ese objeto.

```
def __additional_cards(self, player):
    while not player.is_busted() and player.is_hitting():
        self.deck.deal([player])
        print(player)
        if player.is_busted():
            player.bust()
```

d) El método play().

El método `play()` es donde se define el bucle principal del juego, y guarda una extraordinaria semejanza con el pseudocódigo que escribimos al principio:

```
def play(self):
    # deal initial 2 cards to everyone
    self.deck.deal(self.players + [self.dealer], per_hand = 2)
    self.dealer.flip_first_card()    # hide dealer's first card
    for player in self.players:
        print(player)
    print(self.dealer)

    # deal additional cards to players
    for player in self.players:
        self.__additional_cards(player)

    self.dealer.flip_first_card()    # reveal dealer's first

    if not self.still_playing:
        # since all players have busted, just show the dealer's hand
        print(self.dealer)
    else:
        # deal additional cards to dealer
        print(self.dealer)
        self.__additional_cards(self.dealer)

        if self.dealer.is_busted():
            # everyone still playing wins
            for player in self.still_playing:
                player.win()
        else:
            # compare each player still playing to dealer
            for player in self.still_playing:
                if player.total > self.dealer.total:
                    player.win()
                elif player.total < self.dealer.total:
                    player.lose()
                else:
                    player.push()

    # remove everyone's cards
    for player in self.players:
        player.clear()
    self.dealer.clear()
```

Se reparten las dos cartas iniciales al ordenador y cada uno de los jugadores. Se le da la vuelta a la primera carta del ordenador para ocultar su valor. A continuación, se le dan cartas adicionales a cada jugador siempre que el jugador así lo requiera y no se haya pasado. Si todos los jugadores se han pasado, se gira la primera carta del ordenador y se muestra su puntuación. En caso contrario, el juego continúa. El ordenador recibe más cartas siempre que el total de puntos de su mano sea menor que 17. Si el ordenador se pasa, el resto de jugadores que quedan en la partida ganan. En otro caso, la puntuación de cada jugador

se compara con la puntuación del ordenador. Si la puntuación total del jugador es mayor que la del ordenador, el jugador gana. Si la puntuación total del jugador es menor, el jugador pierde. Si los dos totales son iguales, el jugador empata.

La función main().

La función `main()` obtiene los nombres de todos los jugadores, los pone en una lista, y crea un objeto `BJ_Game` usando la lista como argumento. A continuación, la función llama al mátodo `play()` del objeto `BJ_Game`, y continúa haciéndolo hasta que los jugadores ya no quieran jugar más.

```
def main():
    print("\t\tWelcome to Blackjack!\n")

    names = []
    number = games.ask_number("How many players? (1-7): ", low = 1, high = 8)
    for i in range(number):
        name = input("Enter player name: ")
        names.append(name)
    print()

    game = BJ_Game(names)

    again = None
    while again != "n":
        game.play()
        again = games.ask_yes_no("\nDo you want to play again?: ")
```

Por su parte, el programa principal se limita a llamar a la función `main()` para iniciar el juego:

```
main()
```

9.13. EJERCICIOS DEL CAPÍTULO 9.

Ejercicio 9.1. Crea una clase llamada `Restaurant`. El método `__init__()` debería crear dos atributos: `restaurant_name` y `cuisine_type`. Escribe un método `describe_restaurant()` que imprima estos atributos, y otro método llamado `open_restaurant()` que imprima un mensaje diciendo que el restaurante está abierto. A continuación, crea tres instancias de esa clase (correspondiéndose a tres restaurantes distintos), y llama a sus métodos para obtener su información y dejarlos abiertos. Guarda el programa como `Ejer9.1a.py`.

Copia el programa previo y guárdalo como `Ejer9.1b.py`. A continuación, añade un atributo llamado `number_served` con un valor por defecto de 0. Crea una instancia llamada `restaurant` para esta nueva clase. Imprime el número de clientes que ha servido el restaurante, y luego cambia su valor e imprímelo otra vez. Añade un método llamado `set_number_served()` que permita fijar el número de clientes que han sido servidos. Llama a este método con un nuevo número, e imprime el valor nuevamente. Añade un método llamado `increment_number_served()` que permita incrementar el número de clientes que han sido servidos. Llama a este método un valor de incremento cualquiera, e imprime el valor para comprobar su correcto funcionamiento. Guarda el programa como `Ejer9.1b.py`.

Ejercicio 9.2. Escribe una clase llamada `User`. Crea dos atributos llamados `name` y `surname`, y a continuación unos cuantos atributos más que almacenen la información típica del perfil de un usuario. Crea un método llamado `describe_user()` que imprima un resumen de la información de usuario. Haz otro método llamado `greet_user()` que imprima una saludo personalizado para el usuario. Por último, crea

varias instancias que representen diferentes usuarios, y llama a ambos métodos para cada usuario. Guarda el programa como `Ejer9.2a.py`.

Copia el programa previo y guárdalo como `Ejer9.2b.py`. Ahora, añade un atributo llamado `login_attemps`, y escribe un método `increment_login_attemps()` que incremente el valor de `login_attemps` en 1 unidad. Escribe otro método llamado `reset_login_attemps()` que resetee el valor de `login_attemps` a 0. Crea una nueva instancia de `User` y llama al método `increment_login_attemps()` varias veces. Imprime el valor de `login_attemps` y asegúrate de que se ha incrementado adecuadamente. Finalmente, llama al método `reset_login_attemps()` y comprueba que ha reseteado el valor de `login_attemps` a 0. Guarda el programa como `Ejer9.2b.py`.

Ejercicio 9.3. Herencia.
(a) Un puesto de helados es un tipo específico de restaurante. Escribe una clase llamada `IceCreamStand` que herede de la clase `Restaurant` que escribiste en el ejercicio 9.1. Añade un atributo llamado `flavors` que almacene una lista de sabores de helados. Escribe un método que muestre esta lista de sabores. Finalmente, en el programa principal, crea una instancia de esta clase y llama al método para comprobar su funcionamiento. Guarda el programa como `Ejer9.3a.py`.

(b) Un administrador es un tipo especial de usuario. Escribe una clase llamada `Admin` que herede de la clase `User` que escribiste en el ejercicio 9.2. Añade un atributo `privileges` que almacene una lista de cadenas como `"can add post"`, `"can delete post"`, `"can ban user"`, etc. Escribe un método llamado `show_privileges()` que liste el conjunto de privilegios del administrador. Prueba el correcto funcionamiento de la clase y del método en el programa principal. Guarda el programa como `Ejer9.3b.py`.

(c) Escribe una clase `Privileges` independiente. La clase debería tener un atributo, `privileges`, que almacene una lista de cadenas como la del apartado previo. Mueve el método `show_privileges()` de la clase `Admin` a la clase `Privileges`. Ahora, crea un atributo en la clase `Admin` que sea un objeto de la clase `Privileges`. En el programa principal, crea una instancia de `Admin` y usa el método de su atributo `Privileges` para mostrar sus privilegios. Guarda el programa como `Ejer9.3c.py`.

Ejercicio 9.4. Dado.
Escribe una clase `Die` (dado) con un atributo llamado `sides` (caras), que tenga un valor por defecto de 6. Escribe un método llamado `roll_die()` que imprima la cara que sale aleatoriamente al tirar el dado. Construye un dado de 6 caras y tíralo 10 veces. A continuación, escribe un dado de 10 caras y otro de 20 caras, y tíralos 10 veces. Guarda el programa como `Ejer9.4.py`.

Ejercicio 9.5. Crea una clase llamada `Triangle`. Su método constructor debería tomar como argumentos `self`, `angle1`, y `angle2`. Crea un método llamado `find_angle3()` que calcule el valor del tercer ángulo a partir de los dos primeros. Crea una variable llamada `my_triangle` que almacene una nueva instancia de la clase triángulo, e imprime los valores de `angle1`, `angle2`, y `angle3`. Define una clase `Equilateral` que derive de la clase `Triangle`, y que sobrecargue los métodos que creas necesario para producir esta clase. Guarda el programa como `Ejer9.5.py`.

Ejercicio 9.6. En la sección 9.3 creamos la clase `Car`. Un coche eléctrico es un tipo especial de coche. Escribe una clase `ElectricCar` que derive de la clase `Car`. La clase `ElectricCar` heredará todos los atributos y métodos que ya escribimos para la clase `Car`. El método constructor de los objetos `ElectricCar` llamará al constructor de su clase base `Car` con los mismos parámetros (`make`, `model`, `year`), pero también creará un nuevo atributo, a saber, un objeto `battery` de una nueva clase `Battery`, que representará su batería.

La clase `Battery` representa la batería de un objeto `ElectricCar`. Esta clase necesita un constructor que inicialice el atributo `battery_size`, el cual empieza por defecto a $70\,kWh$ a no ser que se le especifique otro valor.

Battery también dispondrá de un método `__str__()` que permita mostrar por pantalla el tamaño de la batería que dispone un cierto coche eléctrico, mediante un mensaje similar a `This electric car has a 70-kWh battery`.

También tendrá un método `get_range()` que le permite indicar el kilometraje máximo que podemos hacer con la batería a plena carga. El kilometraje posible será de $140\,km$ para baterías menores o iguales a $70\,kWh$, de 185 km para baterías desde $70\,kWh$ y hasta $90\,kWh$, y de $220\,km$ para baterías superiores a $90\,kWh$.

Por último, `Battery` disponde de un método `change_battery()` que le permite cambiar la batería por otra nueva (posiblemente con un nuevo tamaño). Este método recibe como argumento el tamaño de la nueva batería.

En el programa principal, instancia un objeto `Car` y otro objeto `ElectricCar`. Añade las instrucciones necesarias para leer, actualizar, e incrementar el odómetro de ambos coches. Añade también el código necesario para mostrar por pantalla los datos de ambos coches, obtener el kilometraje máximo alcanzable del coche eléctrico, cambiar la batería del coche eléctrico, y obtener el nuevo kilometraje máximo con esa nueva batería.

Por último, sobrecarga los métodos heredados `update_odometer()` e `increment_odometer()` del coche eléctrico para no permitirle pasar del kilometraje máximo que permite su batería actual.

AMPLIACIÓN:

(a) Crea un método llamado `charge_battery()` que permita cargar la batería. Al cargar la batería, el coche eléctriuco podrá volver a recorrer todo el kilometraje que le permite su batería recién cargada. Este método deberá actuar sobre los métodos heredados `update_odometer()` e `increment_odometer()`, para controlar el nuevo kilometraje máximo posible tras haber cargado la batería. Por ejemplo, un coche con una batería de $80\,kWh$ puede hacer un máximo de $140\,km$. Pero si ya ha recorrido $100\,km$ y carga su batería, podrá recorrer otros $140\,km$, hasta un máximo de $240\,km$.

(b) Al cambiar su batería, el coche dispondrá de kilometraje extra que podrá recorrer sobre el kilometraje que ya había hecho. Por consiguiente, el método `change_battery()` también tendrá consecuencias sobre los métodos heredados `update_odometer()` e `increment_odometer()`. Por ejemplo, si un coche de $70\,kWh$ ha recorrido $140\,km$, ya no podrá recorrer más, pero si cambiamos a una batería de $100\,kWh$, podrá recorrer otros $220\,km$ extra.

Guarda el programa como `Ejer9.6.py`.

Ejercicio 9.7. Escribe un programa que simule una televisión, creándola como un objeto. El usuario debe ser capaz de introducir un número de canal y subir o bajar el volumen. Asegúrate de que el número de canal y el nivel de volumen siempre permanecen dentro de unos márgenes válidos. Guarda el programa como `Ejer9.7.py`.

Ejercicio 9.8. Mejora el programa de la mascota virtual escrito en la sección 9.7 para permitir que el usuario pueda especificar cuánta comida le da a la mascota, y durante cuánto tiempo juega con ella. Haz que estos valores afecten a lo rápido que caen los niveles de apetito y de aburrimiento de la mascota. Guarda el programa como `Ejer9.8.py`.

Ejercicio 9.9. Hackea el programa de la mascota virtual de la sección 9.7 de forma que muestre los valores exactos de los atributos del objeto. Esto debes hacerlo imprimiendo el objeto cuando el usuario seleccione una opción secreta, que no se muestra en el menú. Pista: Añade un método especial `__str__()` a la clase `Critter`. Guarda el programa como `Ejer9.9.py`.

Ejercicio 9.10. Escribe un programa llamado `Critter Farm` (granja de mascotas) instanciando varios objetos `Critter` y siguiéndoles la pista mediante una lista. Reutiliza el programa de la sección 9.7, pero en lugar de requerir que el usuario solo cuide de una mascota, oblígalo a cuidar de toda una granja de mascotas. Cada elección del menú debería permitirle al usuario realizar la misma acción sobre todas las mascotas (alimentar a todas las mascotas, jugar con todas las mascotas, o escuchar a todas las mascotas). Para hacer el programa más interesante, dale a cada mascota unos niveles iniciales de apetito y aburrimiento aleatorios. Guarda el programa como `Ejer9.10.py`.

Ejercicio 9.11. En dos dimensiones y en coordenadas Cartesianas, un punto viene definido por dos números x e y que pueden tratarse colectivamente como un solo objeto. Los puntos suelen escribirse entre paréntesis, separando mediante una coma las dos coordenadas. Por ejemplo, $(0,0)$ representa el origen, y (x,y) representa el punto en la coordenada horizontal x y la coordenada vertical y (siendo x e y dos números cualesquiera)

(a) Escribe una clase llamada `Point` con dos atributos `x` e `y`, cuyo método constructor cree un punto en las coordenadas x e y proporcionadas en la llamada al método (parámetros `initX` e `initY`).

(b) A continuación, escribe dos métodos sencillos para la clase `Point` que nos permitan obtener la coordenada x del punto, método `getX()`, y la coordenada y del punto, método `getY()`.

(c) Crea un método para la clase `Point` llamado `distance_from_origin()` que obtenga la distancia del punto al origen, a saber, $r = \sqrt{x^2 + y^2}$.

(d) A continuación, crea una *función* llamado `distance()` que calcule la distancia entre dos puntos (x_1, y_1) y (x_2, y_2):

$$x_d = x_2 - x_1$$

$$y_d = y_2 - y_1$$

$$d = \sqrt{x_d^2 - y_d^2}$$

(e) Ahora escribe un método `__str__()` que permita obtener la representación en forma de cadena de un punto, como por ejemplo, `"(2,-3)"`, para imprimirlo por pantalla.

(f) Crea un método llamado `halfway()` que encuentre el punto medio entre el punto actual y otro punto objetivo. Este método recibirá como argumento un objeto `Point` con el punto objetivo, y devolverá un objeto `Point` con las coordenadas del punto medio buscado.

(g) Escribe un método `distance_from_point()` que funcione de forma similar a `distance_from_origin()`, excepto que ahora el método recibe un objeto `Point` como parámetro, y calcula la distancia entre el punto actual y el punto pasado como argumento.

(h) Crea un método `reflect_x()` que devuelva un nuevo objeto `Point`, que será la reflexión del punto actual respecto al eje x. Por ejemplo, si el punto es $(3,5)$, el método debe devolver $(3,-5)$.

(i) Añade un método `slope_from_origin()` que devuelva la pendiente (slope) de la línea que conecta el origen con el punto actual. De matemáticas de 3º de ESO recordarás que, si las coordenadas del punto son x e y, la pendiente de la recta desde el origen a ese punto es:

$$m = \frac{y}{x}$$

¿En qué casos podrá fallar este método? Devuelve `None` en aquellos casos que den problemas.

(j) La ecuación de una línea recta es:

$$y = mx + b$$

, donde m es la pendiente y b el "valor de la ordenada en el origen" (coordenada y del punto de cruce de la recta con el eje vertical y). Escribe un método para la clase `Point` que reciba como arumento un segundo objeto `Point`, y que compute la ecuación de la línea recta que une esos dos puntos. El método debe devolver una tupla con los dos coeficientes m y b de la recta. De matemáticas de 3º de ESO recordarás que, si las coordenadas de los dos puntos son (x_1, y_1) y (x_2, y_2), la pendiente es:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

, y de la ecuación punto - pendiente de la recta:

$$y - y_1 = m(x - x_1) \qquad \text{ó} \qquad y - y_2 = m(x - x_2)$$

, obtenemos que:

$$y = mx - (mx_1 - y_1) \qquad \text{ó} \qquad y = mx - (mx_2 - y_2)$$

, donde:

$$b = -(mx_1 - y_1) \qquad \text{ó} \qquad b = -(mx_2 - y_2)$$

(k) Escribe un método llamado `move()` que reciba dos parámetros, `dx` y `dy`. El método hará que el objeto Point se mueva en la sdirecciones x e y el número de unidades dadas por `dx` y `dy`.

Guarda el programa como `Ejer9.11.py`.

Ejercicio 9.12. Fracciones.
Vamos a crear una clase `Fraction` que incluya algunos de los métodos más comunes para trabajar y operar con fracciones. Una fracción puede verse como una pareja de enteros, llamados numerador y denominador, separados por una línea de fracción. Con ello en mente:

(a) Implementa un método constructor que le permita al usuario proporcionar el numerador (parámetro `top`) y el denominador (parámetro `bottom`).

(b) Escribe un método `__str__()` que permita obtener la representación en forma de cadena de la fracciónpara imprimir la fracción por pantalla, como por ejemplo `"3/5"`.

(c) Escribe sendos métodos, `getNum()` y `getDen()` para obtener el numerador y el denominador del objeto `Fraction`.

(d) Escribe una *función* que obtenga el máximo común dividor de dos enteros, `gcd()`.

(e) Una vez disponemos de una función que calcule el máximo común divisor, crea un método `simplify()` que reciba el numerador y el denominador de un objeto `Fraction`, y que simplifique la fracción usando el

máximo común divisor. Por ejemplo, si la fracción es $12/16$, el máximo común divisor es 4, y la fracción simplificada queda como $3/4$.

(f) Recordemos que sólo se pueden sumar fracciones con el mismo denominador. Para sumar dos fracciones con distinto denominador, hemos de hallar el denominador común, y ajustar numeradores. Una forma sencilla de hacer esto es:

$$\frac{a}{b}+\frac{c}{d}=\frac{(ad+cb)}{bd}$$

Escribe un método `add()` que reciba como argumento un objeto `Fraction` con la fracción a sumar a la fracción actual, y que devuelva un objeto `Fraction` que represente la fracción suma adecuadamente simplificada.

(g) Implementa los métodos necesarios para restar, multiplicar, y dividir dos fracciones. Estos métodos deben devolver las fracciones resultados de estas operaciones debidamente simplificadas.

Guarda el programa como `Ejer9.12.py`.

Ejercicio 9.13. Rectángulos.
Podemos representar un rectángulo conociendo tres datos: las coordenadas del punto en la esquina inferior izquierda (por ejemplo, el punto $(4,5)$), su anchura (por ejemplo, 6), y su altura (por ejemplo, 5). Pista: Para todo el ejercicio, es recomendable importar y usar la clase `Point`.

(a) Crea una clase `Rectangle` usando esta idea.

(b) Añade los métodos `getWidth()`, `getHeight()`, y `__str__()` a la clase `Rectangle`.

(c) Escibe un método `area()` a la clase `Rectangle` que devuelva el área de un objeto rectángulo.

(d) Escribe un método `perimeter()` que obtenga el perímetro de un objeto rectángulo.

(e) Escribe un método `traspose()` que intercambie la altura y la anchura de un objeto rectángulo.

(f) Escribe un método `diagonal()` que devuelva la longitud y la ecuación $y=mx+b$ de la recta diagonal que va desde la esquina inferior izquierda a la esquina superior derecha de un objeto rectángulo.

(g) Escribe un nuevo método `contains()` que compruebe si un punto cae dentro de un objeto rectángulo.

AMPLIACIÓN: En los videojuegos suele usarse una "caja delimitadora" rectangular alrededor de los personajes para detectar colisiones entre ellos. Por ejemplo, para decidir si un proyectil ha alcanzado a una nave espacial, se mira si sus dos rectángulos delimitadores se solapan en algún lado. Usando los resultados del apartado previo, escribe una función que determine si dos rectángulos colisionan. Pista: Piensa en todos los casos en los que dos rectángulos pueden colisionar.

Guarda el programa como `Ejer9.13.py`.

Ejercicio 9.14. Cohetes.
Vamos a escribir una clase `Rocket` para modelar el comportamiento de un cohete. El cohete vendrá descrito por la posición instantánea en la que se encuentra, mediante las coordenadas Cartesianas x e y.

(a) Define una clase `Rocket`, cuyo método constructor permita inicializar el valor de las coordenadas x e y a $(0,0)$. (Las coordenadas $(0,0)$ identifican la posición en Tierra del lugar de lanzamiento).

(b) Escribe dos métodos, `getX()` y `getY()`, que permitan obtener las coordenadas instantáneas x e y de cada objeto cohete que hayamos instanciado.

(d) Escribe un método `__str__()` que permita imprimir las coordenadas actuales del objeto cohete.

(c) Escribe un método `move_up()` para mover al cohete hacia arriba, en el número de unidades *enteras* y *positivas* indicadas como argumento. En base a este método, escribe un método `launch()` que pemita hacer despegar al cohete. Este método imprime por pantalla un mensaje informativo, y hace subir al cohete hasta la altura indicada como argumento en el método. El método permite cambiar el valor booleano de un atributo `is_flying`, que nos indica si un cohete ha despegado ya o todavía está en tierra. (Este atributo te obigará a modificar el método constructor).

(f) Define un método general `move()` que te permita mover un cohete que ya haya despegado en una dirección cualquiera, pasándole como argumentos los valores del desplazamient horizontal (positivo o negativo) y del desplazamient vertical (positivo o negativo). Asegúrate de que este método no funcione con cohetes que no hayan despegado, mostrando el correspondiente mensaje de error. Asegúrate también de que el movimiento del cohete no le hace caer de nuevo a tierra.

(g) Define un método `land()` que permita a un cohete en vuelo aterrizar en la posición horizontal en la que se encuentre en ese momento. Al aterrizar, este método debe fijar correctamente el valor del atributo `is_flying`.

(h) Modifica el método constructor para permitir crear objetos cohete en tierra ($y = 0$), pero en una posición horizontal indicada por el usuario.

(i) Crea toda una flota de cohetes (en distintas posiciones horizontales), añádelos a una lista de objetos cohete, e imprime sus posiciones para visualizar dónde comienzan. Hazlos despegar, muévelos por el cielo, e imprime nuevamente sus respectivas posiciones tras haberlos movido. Hazlos aterrizar e imprime sus posiciones finales. Constata que se trata de objetos totalmente independientes.

(j) Crea un método llamado `get_distances()` que reciba una lista de cohetes como argumento. Este método le permite a un objeto cohete calcular la distancia a la que se encuentra actualmente respecto a todos los cohetes de la lista pasada como argumento. El método debe devolver una lista de distancias. Además, si la distancia a otro cohete es demasiado corta (digamos, 2 unidades o menos), el método debe imprimir por pantalla un mensaje de advertencia. Si alguna de las distancias es cero, el método debe imprimir un mensaje informando de la colisión. *Cuidado*: No calcules la distancia del cohete respecto a sí mismo, porque siempre será cero, y siempre detectará una colisión.

(k) Un cohete Soyuz es un tipo especial de cohete. El cohete Soyuz es un cohete ruso de 3 etapas usado para llevar a los astronautas a la Estación Espacial Internacional (EEI). Crea una clase `Soyuz` que derive de la clase `Rocket`. Esta nueva clase heredará todos los atributos y métodos de su clase base, pero se diferenciará de ella en lo siguiente:

- La clase `Soyuz` debe añadir los siguientes atributos: número de fases actuales (empieza con tres por defecto, y poco a poco las irá perdiendo), número de vuelos realizados, y número máximo de vuelos permitidos.
- La clase Soyuz debe llevar la cuenta del número de vuelos que ha realizado. Sobrecarga y/o modifica los métodos que creas necesarios para hacerlo.
- Crea un nuevo método para hacer despegar a la Soyuz con destino a la EEI, que está a una altura de 400 unidades de distancia vertical. El vuelo vertical es progresivo, digamos, en pasos de 10. Ve mostrando por pantalla la localización instantánea del cohete. Este método sólo funciona si la Soyuz no ha superado el número máximo de vuelos. Para despegar, activa los propulsores de su primera etapa, hecho que indica mostrando un mensaje por pantalla. Al llegar una altura de 100, suelta su primera etapa, y muestra un mensaje diciendo que ha soltado su primera etapa, y que activa los

propulsores de la segunda etapa. (Actualiza adecuadamente el valor de su atributo de número de fases). Al llegar a una altura de 250, muestra un mensaje que indica que suelta la segunda etapa, y que activa los propulsores de la tercera etapa. Al llegar a los 400, muestra un mensaje indicando que ha atracado en la EEI. Actualiza los valores de los atributos que creas convenientes.

- Crea un nuevo método para hacer volver a la Soyuz desde la EEI a la Tierra. Para ello, haz descender a la Soyuz progresivamente (digamos, en pasos de 10). Ve mostrando por pantalla la localización instantánea del cohete. A una altura de 200, abandonamos la tercera fase, y activamos los propulsores del módulo de mando. (Actualiza adecuadamente el valor de su atributo de número de fases). A los 100, la fricción con la atmósfera nos obliga a activar los escudos términos. A los 30, abrimos los paracaídas para ralentizar la caída. Actualiza los valores de los atributos que creas convenientes.

Guarda el programa como `Ejer9.14.py`.

Ejercicio 9.15. Escribe una versión con una sola carta para el juego de la guerra, en el que cada jugador recibe una única carta, y donde gana el jugador con la carta más alta. (El orden de valores de las cartas es: as, 2, 3, …, 10, J, Q, K). Guarda el programa como `Ejer9.15.py`.

Ejercicio 9.16. Mejora el proyecto del juego de blackjack permitiendo que los jugadores apuesten. Sigue la pista de los fondos disponibles para cada jugador, y elimina de la partida a los jugadores que se queden sin dinero. Guarda el programa como `Ejer9.16.py`.

Ejercicio 9.17. Añade el código para controlar errores en el juego de blackjack. Por ejemplo, antes de que empiece una nueva ronda, asegúrate de que el mazo dispone de las cartas suficientes. Si no es así, rellénalo de nuevo y barájalo. Encuentra otros posibles errores que podrías comprobar, y añade el código necesario para evitarlos. Guarda el programa como `Ejer9.17.py`.

Ejercicio 9.18. Crea un juego de aventuras sencillo usando objetos, en el que un jugador pueda viajar entre distintas localizaciones interconectadas. Guarda el programa como `Ejer9.18.py`.

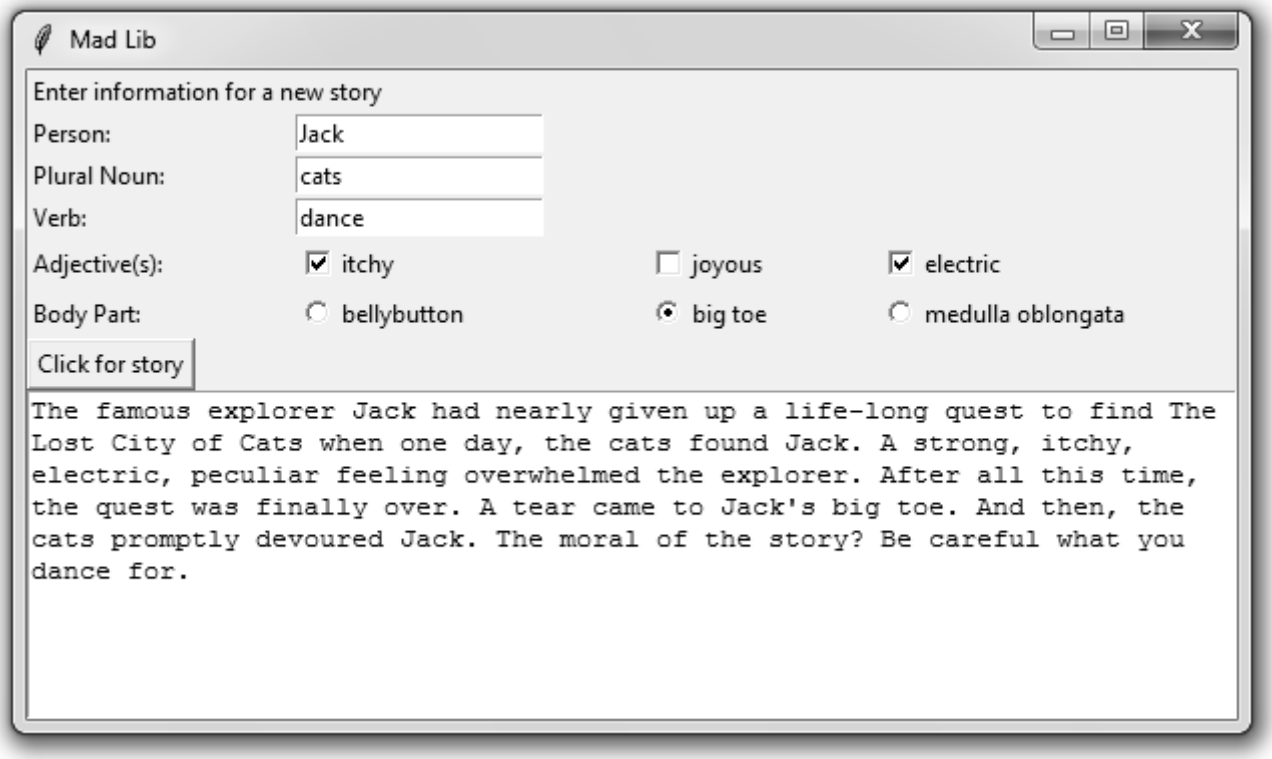
10. INTERFACES GRÁFICAS DE USUARIO (GUIs).

10.1. INTRODUCCIÓN.

Hasta ahora, todos nuestros programas han usado texto plano para interactuar con el usuario. Pero hay formas más sofisticadas de presentar y aceptar la información. Una **interfaz gráfica de usuario** (**GUI**, Graphic User Interface) proporciona un mecanismo visual para que un usuario interactúe con el ordenador. Los sistemas operativos domésticos más populares usan una GUI, permitiendo que las interacciones usuario-ordenador sean más simples y consistentes. En este capítulo aprenderemos a crear estas interfaces gráficas.

10.2. ANÁLISIS DE UNA GUI.

El programa que escribiremos como proyecto final de este capítulo es un programa tipo Mad Lib, como el que desarrollamos en el ejercicio 8.22. Este programa le pide al usuario que le ayude a crear una historia. El usuario proporciona el nombre de una persona, un sustantivo en plural, y un verbo. El usuario también puede elegir varios adjetivos, y seleccionar una de entre varias partes del cuerpo. El programa recibe toda esta información y crea la historia. Y todo esto lo hace comunicándose con el usuario a través de la GUI que mostramos en la figura:



En esta GUI podemos ver la ventana o "frame" que incluye a todos los demás elementos del GUI, etiquetas de texto o "labels" (Enter information for a new story), entradas para texto o "text entries" (Person, Plural Noun, y Verb), botones o "buttons" (Click for story), casillas de verificación o "check buttons" (itchy, joyous, y electric), botones de opción o "radio buttons" (bellybutton, big toe, y medula oblongata), y cajas de texto o "text boxes" (donde se imprime la historia).

Para crear una GUI con Python, necesitamos usar una toolkit para GUIs. Hay muchas toolkits para interfaces gráficas en Python, pero la más popular es Tkinter, y esta seá la que usaremos en este capítulo.

NOTA: En sistemas operativos distintos de Windows, puede ser necesario descargar e instalar software adicional para poder usa rla toolkit Tkinter. Para más información, visitar http://www.python.org/topics/tkinter.

Para crear elementos de interfaz de usuario debemos instanciar objetos a partir de las clases del módulo `tkinter`, que es parte de la toolkit Tkinter. La tabla a continuación describe cada elemento GUI presente en la interfaz de usuario que crearemos para nuestra Mad Lib, e indica su clase correspondiente:

SELECTED GUI ELEMENTS

Element	tkinter	Class Description
Frame	`Frame`	Holds other GUI elements
Label	`Label`	Displays uneditable text or icons
Button	`Button`	Performs an action when the user activates it
Text entry	`Entry`	Accepts and displays one line of text
Text box	`Text`	Accepts and displays multiple lines of text
Check button	`Checkbutton`	Allows the user to select or not select an option
Radio button	`Radiobutton`	Allows, as a group, the user to select one option from several

10.3. PROGRAMACIÓN DIRIGIDA POR EVENTOS.

Los programas GUI están dirigidos por eventos, lo que significa que responden a la ocurrencia de eventos, independientemente del orden en el que se produzcan. La **programación dirigida por eventos** es algo diferente a la programación que hemos estado empleando hasta ahora.

Para entender cómo funciona la programación dirigida por eventos, tomemos como ejemplo el proyecto Mad Lib de este capítulo. Si fuésemos a escribir un programa similar con nuestros actuales conocimientos de Python, probablemente le formularíamos al usuario una serie de preguntas con la función `input()`. Por ejemplo, le pediríamos el nombre de una persona, un sustantivo en plural, un vrbo, etc. Como resultado, el usuario debería de proporcionar cada uno de estos datos en orden. Pero si escribimos el código empleando programación dirigida por eventos, como hacemos al programar una GUI, el usuario podría introducir la información en cualquier orden. Además, el instante en el que el programa genera la historia también lo decidiría el usuario (al clicar en el botón).

Cuando escribimos programas dirigidos por eventos, *asociamos* **eventos** (esto es, cosas que pueden ocurrir con los objetos del programa) con **manejadores de eventos** (esto es, el código que se ejecuta cuando ocurrenesos eventos). A modo de ejemplo concreto, consideremos nuestro futuro programa Mad Lib. Cuando el usuario pinche en el botón "Click for story" (el evento), el programa invocará un método que imprime la historia (el manejador del evento). Para que esto ocurra, debemos *asociar* la acción de pinchar sobre el botón con el método que muestra la historia.

Definiendo todos nuestros objetos, eventos, y manejadores de eventos, podemos establecer cómo funciona nuestro programa. Una vez hecho esto, podemos iniciar el programa insertando un **bucle de evento**, en el que el programa queda a la espera de que ocurran los eventos indicados.

Puede que algunos de estos conceptos no hayan terminado de quedarnos claros. No hay que preocuparse, después de ver unos cuantos ejemplos, todo terminará encajando.

10.4. USAR UNA VENTANA RAÍZ.

El elementos base de nuestro programa GUI es su ventana raíz, sobre la que ubicaremos el resto de elementos de la GUI. Si nos imaginamos la GUI como un árbol, la ventana raíz es, como su propio nombre indica, la raíz. Nuestro árbol puede ramificarse en todas direcciones, pero cada una de sus partes, directamete o indirectamente, provienen de la raíz.

A modo de ejemplo, vamos a crear un programa GUI muy sencillo: una simple ventana. La figura muestra el resultado de este programa:

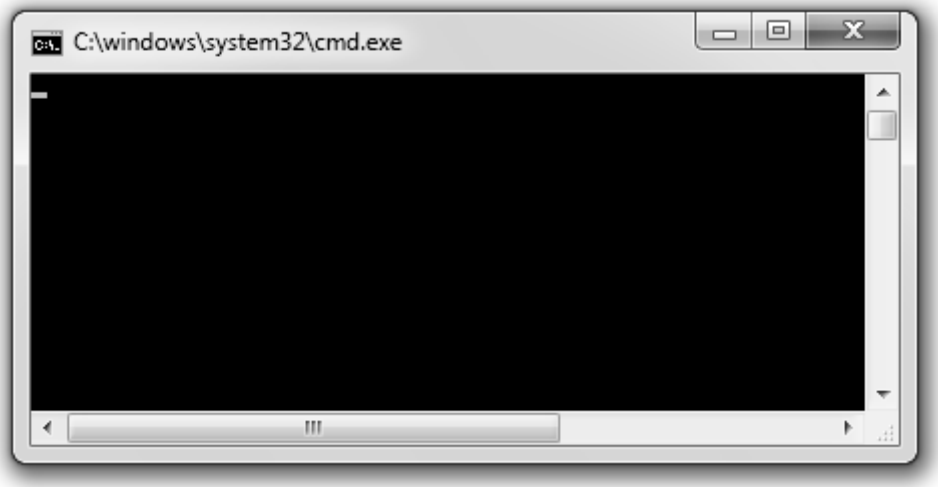
NOTA 1: Al ejecutar un programa Tkinter directamente desde el IDLE, podría bloquearse el programa o el propio IDLE. La solución más sencilla a este problema es ejecutar el programa Tkibter directamente. En Windows, basta con hacer un doble click en el icono del programa.

NOTA 2: Aunque podemos ejecutar un programa Tkinter con un doble click sobre su icono, tendremos problemas si el programa contiene un error: La consola de comandos se cerrará antes de que podamos leer el mensaje de error. Para solventar este contratiempo, en Windows podemos crear un archivo por lotes que ejecute el programa y lo pause después de que el programa termine, manteniendo la consola de comandos abierta para poder ver cualquier mensaje de error. Por ejemplo, si la ruta a nuestro programa es C:\Users\Usuario\Dropbox\PYTHON\PROGRAMAS\simple_gui.py, basta con crear un archivo por lotes `simple_py.bat` que contenga las siguiente líneas:

```
@py.exe C:\Users\Usuario\Dropbox\PYTHON\PROGRAMAS\simple_gui.py %*
pause
```

Una vez escrito el archivo por lotes `simple_py.bat`, podemos ejecutarlo (lo que lanzará la ejecución del programa `simple_gui.py`) haciendo doble click sobre su icono. (Recordar que también podemos utilizar la ventana de ejecución de Windows, WIN + R, para lanzar el programa `simple_gui.py` simplemente escribiendo su nombre. Para ello, la primera línea del archivo `simple_gui.py` debe ser `#! python3`).

Además de la vantana raíz mostrada en la figura, el programa debería generar una ventana de consola de comandos como la mostrada en la figura:



Aunque nos parezca que esta consola de comandos es solo estropea la apariencia de nuestra bonita ventana GUI, no debemos descartarla. La consola de comandos nos proporcionará una valiosa información cuando nuestro programa Tkinter genere errores. Otra razón por la que nunca debemos cerrar la consola de comandos es porque eso también cerraría nuestro programa GUI.

> NOTA 3: Una vez hayamos conseguido que nuestro programa GUI funcione perfectamente (sin errores), puede que queramos dejar de mostrar la consola de comandos que lo acompaña. En Windows, la forma más fácil de hacerlo es cambiar la extensión de nuentro programa `.py` por `.pyw`.

Vamos a ponernos a escribir el código de nuestro primer programa GUI. Abre una nueva ventana del editor de archivos, escribe lo siguiente, y guarda el programa como `simple_gui.py`:

```
#! python3

# Simple GUI
# Demonstrates creating a window

from tkinter import *

# create the root window
root = Tk()

# modify the window
root.title("Simple GUI")
root.geometry("200x100")

# kick off the window's event-loop
root.mainloop()
```

Vamos a analizar este código detalladamente.

IMPORTAR EL MÓDULO TKINTER.

Lo primero que hacemos en nuestro programa GUI es importar el módulo `tkinter`:

```
from tkinter import *
```

Recordemos de la sección 2.5 que esta forma de importar módulos nos evita tener que añadir el nombre del módulo, en este caso `tkinter`, delante de todos los contenidos de dicho módulo. Hasta ahora hemos evitado proceder de esta forma, pero la mayoría de programas tkinter implican muchas referencias a las clases y constantes del módulo `tkinter`, y con ello nos ahorraremos un montón de escritura.

CREAR UNA VENTANA RAÍZ.

Para crear una ventana raíz, instancioamos un objeto de la clase `Tk` del módulo `tkinter`:

```
root = Tk()
```

Como hemos indicado antes, notar que no hemos necesitado escribir `root = tkinter.Tk()`. De hecho, al importar el módulo mediante la instrucción `from tkinter import *`, podemos acceder a cualquier elemento del módulo `tkinter` sin tener que usaar el nombre del módulo.

Un comnetario importante: Sólo podemos tener una vantana raíz en un program Tkinter. Si creamos más de una ventana raíz, el programa quedará bloqueado, ya que ambas ventanas lucharán el control del programa.

MODIFICAR LA VENTANA RAÍZ.

A continuación, modificamos la ventana raíz usando unos cuentos métodos del objeto `root` (objeto perteneciente a la clase `Tk`):

```
root.title("Simple GUI")
root.geometry("200x100")
```

El método `title()` permite establecer el título de la ventana raíz. El único argumento que hay que pasarle es el título a mostrar, en la forma de una cadena. Aquí hemos decidido que el título de la ventana raíz sea `Simpe GUI`.

El método `geometry()` sirve para fijar el tamaño de la ventana raíz, en píxeles. El método recibe una cadena (no valores enteros) que representa la anchura y altura de la ventana, seperadas por el carácter `"x"`. Aquí hemos fijado el tamaño de la ventana a 200 píxeles de ancho por 100 píxeles de alto.

ESCRIBIR EL BUCLE DE EVENTO DE LA VENTANA RAÍZ.

Finalmente, arrancamos el bucle de evento de la ventana raíz invocando el método `mainloop()` del objeto `root`:

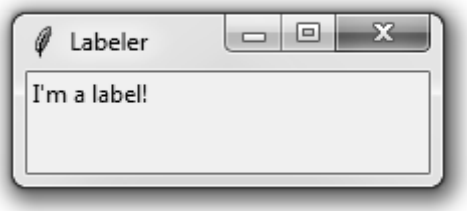
```
root.mainloop()
```

Como resultado, la ventana raíz permanece abierta, a la espera de poder manejar algún evento. Como no hemos definjido eventos que pueda gestionar, la ventana no hace nada. Pero este programa nos ha permitido construir una ventana que podemos mover, redimensionar, minimizar, y cerrar. Haz la prueba: ejecuta el programa y comprueba que funciona como hemos indicado.

10.5. USAR ETIQUETAS.

A los elementos GUI se les suele llamar **widgets** (abreviatura de "window gadgets", o "dispositivos de ventana"). El widget más sencillo es el widget `Label` (**etiqueta**), que consiste simplemente un texto o icono (o ambos) no editable. Un widget `Label` nos permite etiquetar una parte de nuestra GUI. Suele usarse para etiquetar otros widgets. Al contrario que el resto de widgets, las etiquetas no son interactivas. Un usuario no puede pinchar en una etiqueta (bueno, en verdad sí que puede, pero la etiqueta no hará nada en respuesta). Sin embargo, las etiquetas son importantes, y probablemente siempre usaremos algunas en todas nuestras GUIs.

A modo de ejemplo, vamos a escribir un sencillo programa que cree una ventana raíz y le añada una etiqueta. La figura ilustra la salida del programa:



Abre una nueva ventana del editor de archivos, escribe lo siguiente, y guarda el programa como `labeler.py`:

```
#! python3

# Labeler
# Demonstrates a label

from tkinter import *

# create the root window
root = Tk()
root.title("Labeler")
root.geometry("200x50")

# create a frame in the window to hold other widgets
app = Frame(root)
app.grid()

# create a label in the frame
lbl = Label(app, text = "I'm a label!")
lbl.grid()

# kick off the window's event loop
root.mainloop()
```

Vamos a explicar el programa paso a paso:

CONFIGURAR EL PROGRAMA.

Primero, configuramos nuestro programa importando el módulo `tkinter` y creando una ventana raíz:

```
from tkinter import *

# create the root window
root = Tk()
root.title("Labeler")
root.geometry("200x50")
```

CREAR UN MARCO.

Un `Frame` (**marco**) es un elemento que puede alojar otros elementos (como un elemento `Label`). Un marco es como el corcho de un panel de corcho; lo usamos como base sobre la que poner otras cosas. Después de crear la ventana raíz, creamos un marco:

```
# create a frame in the window to hold other widgets
app = Frame(root)
```

Siempre que queramos crear un nuevo widget, debemos pasar el *elemento padre* (esto es, aquella cosa que contendrá a este nuevo elemento) al constructor del nuevo objeto. Aquí, hemos pasado `root` al constructor de `Frame`. Como resultado, se ubicará un nuevo marco dentro de la ventana raíz.

A continuación, invocamos al método `grid()` del nuevo objeto `Frame`:

```
app.grid()
```

`grid()` es un método que todos los widgets tienen disponible. Está asociado a un **gestor de diseño** (layout manager), que nos permite organizar los widgets. Para no complicar las cosas tan pronto, vamos a retrasar la explicación de los gestores de diseño a secciones posteriores.

CREAR UNA ETIQUETA.

Para crear un elemento `Label` debemos instanciar un objeto de la clase `Label`:

```
# create a label in the frame
lbl = Label(app, text = "I'm a label!")
```

Pasándole `app` al constructor del objeto `Label`, le estamos indicando que el marco `app` es el elemento padre del elemento `Label`. Como resultado, la etiqueta se ubica sobre el marco.

A continuación, invocamos al método `grid()` del nuevo objeto `Label`:

```
lbl.grid()
```

Esto asegura que la etiqueta será visible.

ESCRIBIR EL BUCLE DE EVENTO DE LA VENTANA RAÍZ.

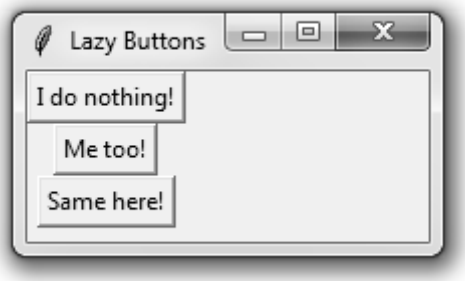
Por último, invocamos el bucle de evento de la vantana raíz para arrancar la GUI:

```
# kick off the window's event loop
root.mainloop()
```

10.6. USAR BOTONES.

Un usuario puede activar un elemento `Button` (**botón**) para realizar alguna acción. Dado que ya sabemos cómo crear etiquetas, aprender a crear botones será muy fácil.

Como ejemplo, vamos a crear un programa GUI con varios botones que no hacen nada cuando el usuario los activa. La figura a continuación ilustra cómo queda el programa:



Escribe el siguiente código en el editor de archivos, y guárdalo como `lazy_buttons.py`:

```
# Lazy Buttons
# Demonstrates creating buttons

from tkinter import *

# create a root window
root = Tk()
root.title("Lazy Buttons")
root.geometry("200x85")
```

```
# create a frame in the window to hold other widgets
app = Frame(root)
app.grid()

# create a button in the frame
bttn1 = Button(app, text = "I do nothing!")
bttn1.grid()

# create a second button in the frame
bttn2 = Button(app)
bttn2.grid()
bttn2.configure(text = "Me too!")

# create a third button in the frame
bttn3 = Button(app)
bttn3.grid()
bttn3["text"]= "Same here!"

# kick off the root window's event loop
root.mainloop()
```

Veamos cómo funciona el programa paso a paso.

CONFIGURAR EL PROGRAMA.

En primer lugar, configuramos el programa importando el módulo `tkinter` y crteando una ventana raíz y un marco:

```
from tkinter import *

# create a root window
root = Tk()
root.title("Lazy Buttons")
root.geometry("200x85")

# create a frame in the window to hold other widgets
app = Frame(root)
app.grid()
```

CREAR BOTONES.

Podemos crear un elemento `Button` instanciando un objeto de la clase `Button`. Eso es lo que hemos hecho en las siguientes líneas del programa:

```
# create a button in the frame
bttn1 = Button(app, text = "I do nothing!")
bttn1.grid()
```

Con este código creamos un nuevo botón con el texto `I do nothing!` sobreimpreso en él. El elemento padre del botón es el marco que creamos antes, lo que significa que le botón se ubica en el marco.

El módulo `tkinter` es muy flexible a la hora de crear, definir, y modificar widgets. Podemos crear un widget y fijar todas sus opciones en una sola línea (como hemos estado haciendo hasta ahora), o podemos crear el widget y fijar o alterar sus opciones más tarde. En el siguiente botón aplicaremos esta estrategia.

En primer lugar, creamos un nuevo botón (el segundo):

```
# create a second button in the frame
bttn2 = Button(app)
bttn2.grid()
```

Notar que ahora el único valor que le hemos pasado un valor al constructor del objeto es `app`, el elemento padre del botón. De esta forma, simplemente hemos añadido al marco un botón en blanco. Sin embargo, también podemos modificar un widget después de haberlo creado, mediante el método `configure()`:

```
bttn2.configure(text = "Me too!")
```

Esta instrucción fija la opción `text` del botón `bttn2` a la cadena `"Me too!"`, lo cual permitirá poner le texto `Me too!` sobre el botón.

Podemos usar el método `configure()` con cualquier opción de cualquier widget. Uncluso podemos usar este método para cambiar una opción que ya hayamos fijado previamente.

A continuación, creamos un tercer botón:

```
# create a third button in the frame
bttn3 = Button(app)
bttn3.grid()
```

, y fijamos su opción `text` usando un método diferente:

```
bttn3["text"]= "Same here!"
```

En esta ocasión accedemos a la opción `text` del botón como si de un diccionario se tratase. Fijamos la opción `text` a la cadena `"Same here!"`, lo que imprime el texto `Same here!` sobre el botón. Cuando fijamos el valor de una opción usando este acceso tipo diccionario, la clave para la opción es el nombre de la opción, en forma de cadena.

ESCRIBIR EL BUCLE DE EVENTO DE LA VENTANA RAÍZ.

Como siempre, con el método `mainloop()` invocamos el bucle de evento de la ventana raíz para arrancar la GUI:

```
# kick off the root window's event loop
root.mainloop()
```

10.7. CREAR UNA GUI USANDO UNA CLASE.

Como aprendimos en el capítulo previo, la utilización de clases permite que el desarrollo de programas sea más fácil de lo que lo sería sin usarlas. A menudo es conveniente escribir los largos programas GUI definiendo nuestras propias clases. En esta sección vamos a aprender cómo escribir programas GUI organizando el código con una clase.

A modo de ejemplo, vamos a reescribir el programa GUI de los botones usando una clase. La pariencia del programa es la misma, pero el código que lo permite estará reestructurado.

Escribe el siguiente código en el editor de archivos, y guárdalo como `lazy_buttons2.py`:

```
#! python3

# Lazy Buttons 2
# Demonstrates using a class with Tkinter

from tkinter import *

class Application(Frame):
    """ A GUI application with three buttons. """
    def __init__(self, master):
        """ Initialize the Frame. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()

    def create_widgets(self):
        """ Create three buttons that do nothing. """
        # create first button
        self.bttn1 = Button(self, text = "I do nothing!")
        self.bttn1.grid()

        # create second button
        self.bttn2 = Button(self)
        self.bttn2.grid()
        self.bttn2.configure(text = "Me too!")

        # create third button
        self.bttn3 = Button(self)
        self.bttn3.grid()
        self.bttn3["text"] = "Same here!"

# main
root = Tk()
root.title("Lazy Buttons 2")
root.geometry("200x85")
app = Application(root)
root.mainloop()
```

Como de costumbre, analizamos detalladamente cada línea del programa.

IMPORTAR EL MÓDULO TKINTER.

Aunque ahora hay importantes cambios estructurales en el programa, todavía debemos importar el módulo `tkinter`:

```
from tkinter import *
```

DEFINIR LA CLASE APPLICATION.

A continuación creamos una nueva clase, llamada `Application`, que está basada en la clase `Frame`:

```
class Application(Frame):
    """ A GUI application with three buttons. """
```

En lugar de instanciar un objeto `Frame`, instanciaremos un objeto `Application` donde albergaremos todos los botones. Esto funcionará correctamente porque un objeto `Application` es solo un tipo especializado de objeto `Frame`.

DEFINIR UN MÉTODO CONSTRUCTOR.

Ahora definimos el constructor de `Application`:

```
def __init__(self, master):
    """ Initialize the Frame. """
    super(Application, self).__init__(master)
    self.grid()
    self.create_widgets()
```

Lo primero que hacemos es llamar al constructor de la superclase. Le pasamos como argumento el elemento padre de `Application`, para que se configure igual que él. Finalmente, invocamos al método `create_widgets()` de `Application`. Este método lo definiremos a continuación.

DEFINIR UN MÉTODO PARA CREAR WIDGETS.

Continuamos definiendo un método `create_widgets()` para crear los tres botones:

```
def create_widgets(self):
    """ Create three buttons that do nothing. """
    # create first button
    self.bttn1 = Button(self, text = "I do nothing!")
    self.bttn1.grid()

    # create second button
    self.bttn2 = Button(self)
    self.bttn2.grid()
    self.bttn2.configure(text = "Me too!")

    # create third button
    self.bttn3 = Button(self)
    self.bttn3.grid()
    self.bttn3["text"] = "Same here!"
```

El código es bastante similar al del programa original. Pero una diferencia importante es que `bttn1`, `bttn2`, y `bttn3` son atributos de un objeto de tipo `Application`. Otra diferencia importante es que usamos `self` como elemento padre de los botones, para indicar que el objeto `Application` es su padre.

CREAR EL OBJETO APPLICATION.

En el programa principal creamos una ventana raíz y le damos un título y un tamaño apropiado:

```
# main
root = Tk()
root.title("Lazy Buttons 2")
root.geometry("200x85")
```

Después instanciamos un objeto `Application` con la ventana raíz como su padre:

```
app = Application(root)
```

Esta instrucción crea un objeto `Application` con la ventana raíz como elemento padre. El constructor del objeto `Application` invoca al método `create_widgets()`. Este método crea los tres botones, con el objeto `Application` como elemento padre.

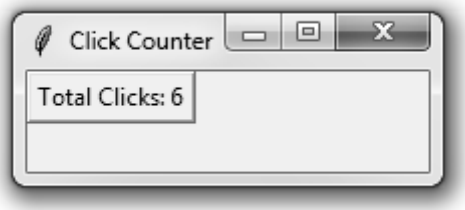
Finalmente, invocamos al bucle de evento de la ventana raíz para lanzar y mantener la ejecuación de la GUI:

```
root.mainloop()
```

10.8. ASOCIAR ELEMENTOS Y MANEJADORES DE EVENTOS.

Los programas GUI que hemos hecho hasta ahora no hacen demasiadas cosas. Esto es debido a que no hemos asociado código a la activación de sus elementos. Ha llegado el momento de darle funcionalidad a nuestra GUI. Esto lo haremos escribiendo manejadores de eventos y asociándolos a eventos propios de los elementos de la GUI.

A modo de ejemplo, vamos a construir una GUI que incorpora un botón que hace algo: Muestra la cuenta del número de veces que ha sido clicado. Técnicamente, el manejador de evento del botón se encargará de llevar la cuenta del número devenes que pinchamos sobre el botón, y de cambiar el texto del botón para mostrarla. La figura muestra la salida del programa:



Escribe el siguiente código en una nueva vantana del editor de archivos, y guárdalo como `click_counter.py`:

```
#! python3

# Click Counter
# Demonstrates binding an event with an event handler

from tkinter import *

class Application(Frame):
    """ GUI application which counts button clicks. """
    def __init__(self, master):
        """ Initialize the frame. """
        super(Application, self).__init__(master)
        self.grid()
        self.bttn_clicks = 0    # the number of button clicks
        self.create_widget()

    def create_widget(self):
        """ Create button which displays number of clicks. """
        self.bttn = Button(self)
        self.bttn["text"]= "Total Clicks: 0"
        self.bttn["command"] = self.update_count
        self.bttn.grid()

    def update_count(self):
        """ Increase click count and display new total. """
        self.bttn_clicks += 1
        self.bttn["text"] = "Total Clicks: " + str(self.bttn_clicks)
```

```
# main
root = Tk()
root.title("Click Counter")
root.geometry("200x50")

app = Application(root)

root.mainloop()
```

Analicemos el programa paso a paso.

CONFIGURAR EL PROGRAMA.

Como siempre, lo primero es importar el módulo `tkinter`:

```
from tkinter import *
```

Después, comenzamos con la definición de la clase `Application`:

```
class Application(Frame):
    """ GUI application which counts button clicks. """
    def __init__(self, master):
        """ Initialize the frame. """
        super(Application, self).__init__(master)
        self.grid()
        self.bttn_clicks = 0    # the number of button clicks
        self.create_widget()
```

Este código ya lo hemos visto antes varias veces. Lo único nuevo es la línea `self.bttn_clicks = 0`, que crea un atributo para llevar la cuenta del número de veces que el usuario princhе en el botón.

ASOCIAR EL MANEJADOR DE EVENTO.

En el método `create_widget()` hemos creado un único botón:

```
def create_widget(self):
    """ Create button which displays number of clicks. """
    self.bttn = Button(self)
    self.bttn["text"]= "Total Clicks: 0"
    self.bttn["command"] = self.update_count
    self.bttn.grid()
```

Ajustamos la opción `command` del elemento `Button` al método `update_count()` que definiremos más adelante. Como resultado, se llama a este método cuando el usuario pincha en el botón. Técnicamente, lo que hemos hecho ha sido asociar un evento (pinchar el el elemento Button) a un manejador de evento (el método `update_count()`).

En general, para asociar la activación de un elemento con un manejadro de evento, debemos fijar la opción `command` del elemento en cuestión.

CREAR EL MANEJADOR DE EVENTO.

A continuación escribimos el método `update_count()`, que se encarga de manejar el evento lanzado cuando el usuario pincha en el botón:

```
def update_count(self):
    """ Increase click count and display new total. """
    self.bttn_clicks += 1
    self.bttn["text"] = "Total Clicks: " + str(self.bttn_clicks)
```

Este método incrementa el número total de toques del botón y cambia el texto del botón para mostrar el nuevo total. Esto es todo lo que debemos hacer para conseguir que un botón haga algo útil.

CONCLUIR EL PROGRAMA.

A estas alturas, el programa principal ya debería resultarnos muy familiar:

```
# main
root = Tk()
root.title("Click Counter")
root.geometry("200x50")

app = Application(root)

root.mainloop()
```

Aquí simplemente creamos la ventana raíz y establecemos su título y dimensiones. Luego instanciamos un nuevo objeto `Application` con la ventana raíz como elemento padre. Finalmente, arrancamos el bucle de evento de la ventana raíz para lanzar la pantalla de la GUI.

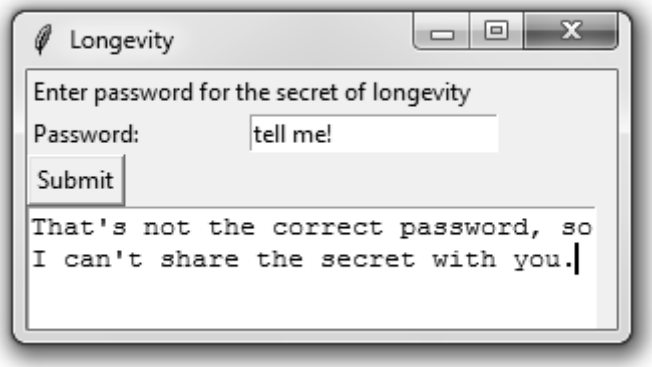
10.9. LOS ELEMENTOS TEXT Y ENTRY, Y EL GESTOR DE DISEÑO GRID.

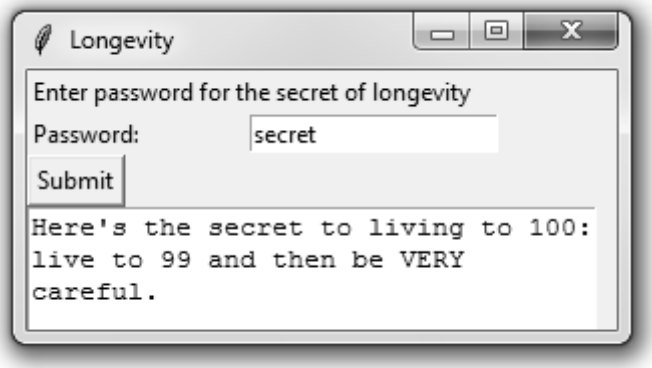
En programación GUI habrá ocasiones en las que querremos que el usuario introduzca un texto. Otras veces querremos mostrar un texto al usuario. En mabos casos, podemos usar widgets de texto. Aquí presentaremos dos de ellos: El elemento `Entry` es útil para una sola línea de texto, mientras que el elemento `Text` se usa para bloques de texto con varias líneas. Podemos leer los contenidos de ambos, y usarlos como elementos para la entrada de datos del usuario. Pero también podemos insetar texto en ellos, y usarlos como elementos para proporcinar información al usuario

Ahora, una vez hemos añadido un conjunto de elementos a un marco, necesitamos una forma de organizarlos. Hasta ahora hemos usado el gestor de diseño `Grid`, pero de manera muy limitada. El gestor de diseño `Grid` nos ofrece un control mucho mayor de la apariencia de nuestra GUI. Este gestor permite ubicar los elementos en localizaciones específicas, tratando al marco como una cuadrícula (grid).

Para aprender a manejar todas estas nuevas funcionalidades, vamos a escribir un programa que revela el secreto para llegar a vivir 100 años. El programa funciona de la siguiente manera: El usuario introduce una contraseña en una entrada de texto, y pncha en el botón de enviar ("submit"). Si la contraseña es correcta, el programa muestra el secreto de la longevidad en una caja de texto.

Las figuras muestran el programa en funcionamiento:





Escribe el siguiente código en el editor de archivos y guarda el programa como `longevity.py`. A continuación, pasaremos a analizar cada línea del programa.

```
#! python3

# Longevity
# Demonstrates text and entry widgets, and the grid layout manager

from tkinter import *

class Application(Frame):
    """ GUI application which can reveal the secret of longevity. """
    def __init__(self, master):
        """ Initialize the frame. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()

    def create_widgets(self):
        """ Create button, text, and entry widgets. """
        # create instruction label
        self.inst_lbl = Label(self, text = "Enter password for the secret" + \
                              " of longevity")
        self.inst_lbl.grid(row = 0, column = 0, columnspan = 2, sticky = W)

        # create label for password
        self.pw_lbl = Label(self, text = "Password: ")
        self.pw_lbl.grid(row = 1, column = 0, sticky = W)

        # create entry widget to accept password
        self.pw_ent = Entry(self)
        self.pw_ent.grid(row = 1, column = 1, sticky = W)
```

```
        # create submit button
        self.submit_bttn = Button(self, text = "Submit", command = self.reveal)
        self.submit_bttn.grid(row = 2, column = 0, sticky = W)

        # create text widget to display message
        self.secret_txt = Text(self, width = 35, height = 5, wrap = WORD)
        self.secret_txt.grid(row = 3, column = 0, columnspan = 2, sticky = W)

    def reveal(self):
        """ Display message based on password. """
        contents = self.pw_ent.get()
        if contents == "secret":
            message = "Here's the secret to living to 100: live to 99 " \
                      "and then be VERY careful."
        else:
            message = "That's not the correct password, so I can't share " \
                      "the secret with you."
        self.secret_txt.delete(0.0, END)
        self.secret_txt.insert(0.0, message)

# main
root = Tk()
root.title("Longevity")
root.geometry("300x150")

app = Application(root)

root.mainloop()
```

CONFIGURAR EL PROGRAMA.

Configuramos nuestro programa como hemos hecho con los últimos programas:

```
from tkinter import *

class Application(Frame):
    """ GUI application which can reveal the secret of longevity. """
    def __init__(self, master):
        """ Initialize the frame. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()
```

Importamos el módulo GUI y comenzamos con la definición de la clase `Application`. En el método constructor, inicializamos el nuevo objeto `Application` asugurándonos de que será visible, e invicamos al método `create_widgets()` del objeto.

UBICAR UN ELEMENTO CON EL GESTOR DE DISEÑO GRID.

A continuación, comenzamos con la definición del método `create_widgets()` y creamos una etiqueta que proporcione instrucciones al usuario:

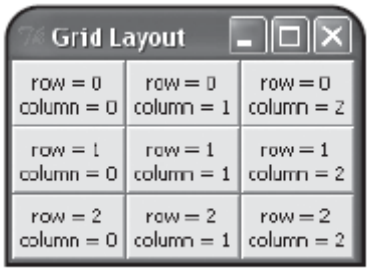
```
def create_widgets(self):
    """ Create button, text, and entry widgets. """
    # create instruction label
    self.inst_lbl = Label(self, text = "Enter password for the secret" + \
                          " of longevity")
```

Nada nuevo hasta ahora. Pero en la siguiente línea, usamos el gestor de diseño `Grid` para espefificar dónde ubicar esta etiqueta:

```
self.inst_lbl.grid(row = 0, column = 0, columnspan = 2, sticky = W)
```

El método `grid()` de un elemento GUI puede recibir valores para muchos parámetros diferentes, pero aquí solo hemos usado cuatro: `row`, `column`, `columnspan`, y `sticky`.

Los parámetros `row` y `column` reciben valores enteros y definen dónde se ubicará un elemento dentro de su elemento padre. En este programa, podemos imaginar que el marco en la ventana raíz es una especie de rejilla, dividida en filas y columnas. Cada intersección de una fila y una columna es una celda en la que podemos ubicar un elemento. La figura la ubicación de nueve elementos `Button` en nueve celdas distintas, usando los números de fila y columna:



Para nuestro elemento `Label` hemos pasado un `0` a row y un `0` a column, lo que ubicará la etiqueta en la esquina suprior izquierda del marco.

Si un widget es muy ancho (como el elemento `Label` que hemos usado en este programa), puede que queramos permitir que se expanda más allá de una celda, sin que ello afecte al correcto espaciado de los otros widgets. El parámetro `columnspan` nos permite extender un widget más de una columna. A este parámetro le hemos pasado un `2` para permitir que nuestra larga etiqueta ocupe dos columnas. Esto significa que la etiqueta ocupará dos celdas consecutivas, una en la fila `0`, columna `0`, y la otra en la fila `0`, columna `1`. (También hay un parámetro `rowspan` para permitir que un widget se extienda más de una fila).

Después de haber establecido qué celda (o celdas) ocupará un elemento, tenemos la posibilidad de alinear el elemento dentro de la celda (o celdas) con el parámetro `sticky`, cuyos valores posibles son direcciones, incluyendo `N`, `S`, `E`, y `W`. El elemento se moverá al cuadrante de la celda (o celdas) que se corresponda con la dirección especificada. Como al parámetro `sticky` del elemento `Label` le hemos pasado el valor `W`, la etiqueta se alineará al oeste (esto es, a la izquierda). Otra forma de decir lo mismo es que la etiqueta está alineada a la izquierda dentro de las celdas que ocupa.

Ahora creamos una etiqueta para la contraseña, que aparecerá en la siguiente fila y alineada a la izquierda:

```
# create label for password
self.pw_lbl = Label(self, text = "Password: ")
self.pw_lbl.grid(row = 1, column = 0, sticky = W)
```

CREAR UN ELEMENTO ENTRY.

Continuamos creando un nuevo tipo de widget: un elemento `Entry`:

```
# create entry widget to accept password
self.pw_ent = Entry(self)
```

Este código crea una entrada de texto en la que el usuario pueda introducir una contraseña.

Ubicamos este elemento de forma que esté en la celda a continuación de la etiqueta de contraseña:

```
self.pw_ent.grid(row = 1, column = 1, sticky = W)
```

Luego, creamos un botón que le permita al usuario enviar la contraseña que ha escrito en la entrada de texto:

```
# create submit button
self.submit_bttn = Button(self, text = "Submit", command = self.reveal)
```

Asociamos la activación del botón al método `reveal()`, que revelará e secreto para la longevidad si el usuario ha introducido la contreseña correcta.

Ubicamos el botón en la siguiente fila, alineado a la izquierda:

```
self.submit_bttn.grid(row = 2, column = 0, sticky = W)
```

CREAR UN ELEMENTO TEXT.

A continuación creamos otro nuevo tipo de widget: un elemento `Text`:

```
# create text widget to display message
self.secret_txt = Text(self, width = 35, height = 5, wrap = WORD)
```

Le pasamos valores a `width` y `height` para fijar las dimensiones de la caja de texto. También le pasamos un valor al parámetro `wrap`, que determina cómo se "empaqueta" el texto en la caja, esto es, qué ocurre cuando el texto llega al extremo derecho de la caja de texto. Los valores posibles para este parámetro son `WORD`, `CHAR`, y `NONE`. El valor `WORD` que hemos usado para nuestro elemento `Text` empaqueta palabras enteras cuando llegamos al extremo derecho de la caja de texto. Esto significa que el cambio de línea no cortaará una palabra por la mitad. El valor `CHAR` empaqueta caracteres, lo que simplemente significa que al llegar al extremo derecho de la caja de texto, el siguiente carácter aparecerá en la línea inferior. `NONE` significa que no se palica empaquetado, y como resultado, solo se puede escribir texto en la primera línea de la caja de texto.

Ahora ubicamos la caja de texto en la siguiente fila, extendiéndose dos columnas:

```
self.secret_txt.grid(row = 3, column = 0, columnspan = 2, sticky = W)
```

OBTENER E INSERTAR TEXTO CON ELEMENTOS DE TEXTO.

Comenzamos con la definición del método `reveal()`, que comprueba si el usuario ha introducido la contraseña correcta. En ese caso, el método muestra el secreto para llegar a vivir `100` años. En caso contrario, el programa le dice al usuario que la contraseña es incorrecta.

Lo primero que hacemoses obtener el texto del elemento `Entry` llamando a su método `get()`:

```
def reveal(self):
    """ Display message based on password. """
    contents = self.pw_ent.get()
```

El método `get()` devuelve el texto almacenado en un elemento de texto. Tanto el elemento `Entry` como el elemento `Text` disponen del método `get()`.

Luego comprobamos si el texto es igual a la cadena "secret", que será nuestra contraseña secreta. Si es así, asignamos a message la cadena que describe el secreto de la longevidad. En caso contrario, asignamos a message la cadena que le indica al usuario que la contraseña que ha introducido no es la correcta:

```
if contents == "secret":
    message = "Here's the secret to living to 100: live to 99 " \
              "and then be VERY careful."
else:
    message = "That's not the correct password, so I can't share " \
              "the secret with you."
```

Ahora que ya tenemos la cadena que queremos mostrar al usuario, debemos insertarla en el elemento Text. En primer lugar, borramos el texto que podría haber en el elemento Text previamente llamando al método delete():

```
self.secret_txt.delete(0.0, END)
```

El método delete() sirve para borrar texto de cualquier elemento de texto. Este método debe recibir un punto de inicio y un punto de final. Para ello, le pasamos números flotantes para representar un número de columna y un número de fila, donde el dígito a la izquierda del punto decimal es el número de fila, y el dígito a la derecha del punto decimal es el número de columna. Por ejemplo, en la línea de código previa, le hemos pasado 0.0 como punto de inicio, lo que significa que el método debería borrar texto comenzando en la fila 0, columna 0, de la caja de texto (esto es, desde el principio de la caja de texto).

tkinter proporciona varias constantes para ayudarnos en el manejo de este método, como por ejemplo END, que indica el final del texto. Por lo tanto, la línea de código previa lo borrará todo desde la primera posición en la caja de texto hasta el final. Tanto el elemento Text como el elemento Entry disponen del método delete().

A continuación insertamos la cadena que queremos mostrar en el elemento Text:

```
self.secret_txt.insert(0.0, message)
```

El método insert() sirve para insertar una cadena en un elemento de texto, como Text o Entry. Este método recibe una posición y una cadena. En la línea de código previa le hemos pasado 0.0 en la posición de isnerción, lo que significa que el método debería empezar a insertar el texto en la fila 0, columna 0 de la caja de texto. También le pasamos message como segundo argumento, para que el mensaje se muestre de forma adecuada en la caja de texto.

> NOTA: El método insert() no reemplaza el texto en el elemento sobre el que lo llamamos, solo lo inserta. Si queremos reemplazar el texto preexistente por un texto nuevo, perimero debemos llamar al método delete().

CONCLUIR EL PROGRAMA.

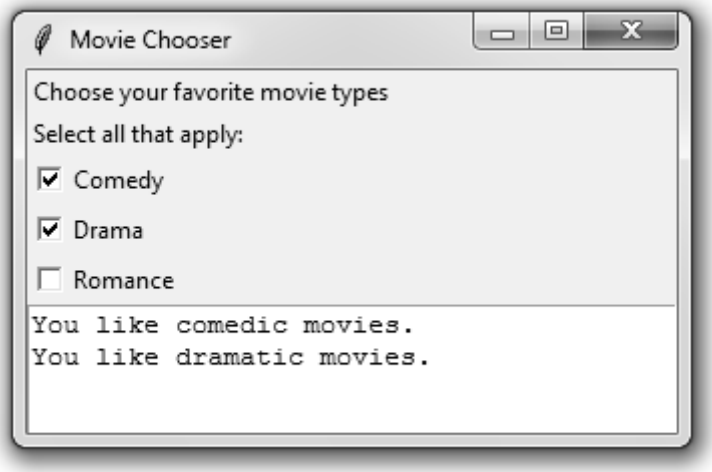
En el programa principal creamos la ventana raíz y fijamos su título y dimensiones. Después creamos un nuevo objeto Application con la ventana raíz como elemento padre. Finalmente arrancamos la aplicación llamando al bucle de evento de la ventana:

```
# main
root = Tk()
root.title("Longevity")
root.geometry("300x150")

app = Application(root)

root.mainloop()
```

10.10. USAR CHECK BUTTONS.

Las **casillas de verificación** (check buttons) le permiten al usuario seleccionar un número cualuiqera de opciones dentro de un grupo de opciones. Aunque esto parece darle al usuario mucha flexibilidad, realmente le permite al programador un mayor control al limitar el conjunto de opciones que el usuario puede elegir a una lista específica.

A modo de ejemplo, vamos a hacer un programa en el que el usuario pueda elegir sus tipos de películas favoritas de una lista de tres: comedia, drama, y romance. Como el programa usa casillas de verificación, el usuario peude seleccionar tantos tipos como desee. La figura muestra el aspecto de esta GUI:



Escribe el siguiente código en el editor de archivos, y guárdalo como `movie_chooser.py`:

```
#! python3

# Movie Chooser
# Demonstrates check buttons

from tkinter import *

class Application(Frame):
    """ GUI Application for favorite movie types. """
    def __init__(self, master):
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()

    def create_widgets(self):
        """ Create widgets for movie type choices. """
        # create description label
        Label(self,
              text = "Choose your favorite movie types"
              ).grid(row = 0, column = 0, sticky = W)

        # create instruction label
        Label(self,
              text = "Select all that apply:"
              ).grid(row = 1, column = 0, sticky = W)
```

```
        # create Comedy check button
        self.likes_comedy = BooleanVar()
        Checkbutton(self,
                    text = "Comedy",
                    variable = self.likes_comedy,
                    command = self.update_text
                    ).grid(row = 2, column = 0, sticky = W)

        # create Drama check button
        self.likes_drama = BooleanVar()
        Checkbutton(self,
                    text = "Drama",
                    variable = self.likes_drama,
                    command = self.update_text
                    ).grid(row = 3, column = 0, sticky = W)

        # create Romance check button
        self.likes_romance = BooleanVar()
        Checkbutton(self,
                    text = "Romance",
                    variable = self.likes_romance,
                    command = self.update_text
                    ).grid(row = 4, column = 0, sticky = W)

        # create text field to display results
        self.results_txt = Text(self, width = 40, height = 5, wrap = WORD)
        self.results_txt.grid(row = 5, column = 0, columnspan = 3)

    def update_text(self):
        """ Update text widget and display user's favorite movie types. """
        likes = ""

        if self.likes_comedy.get():
            likes += "You like comedic movies.\n"

        if self.likes_drama.get():
            likes += "You like dramatic movies.\n"

        if self.likes_romance.get():
            likes += "You like romantic movies."

        self.results_txt.delete(0.0, END)
        self.results_txt.insert(0.0, likes)

# main
root = Tk()
root.title("Movie Chooser")
app = Application(root)
root.mainloop()
```

CONFIGURAR EL PROGRAMA.

Importamos el módulo GUI y comenzamos con la definición de la clase `Application`:

```
from tkinter import *

class Application(Frame):
    """ GUI Application for favorite movie types. """
    def __init__(self, master):
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()
```

PERMITIR A UN ELEMENTO PADRE SER LA ÚNICA REFERENCIA.

Ahora creamos una etiqueta que describa el programa:

```
def create_widgets(self):
    """ Create widgets for movie type choices. """
    # create description label
    Label(self,
          text = "Choose your favorite movie types"
          ).grid(row = 0, column = 0, sticky = W)
```

Sin embargo, hay una diferencia importante entre esta etiqueta y todas las que hemos creado hasta ahora: Aquí no estamos asignando el objeto `Label` resultante a una variable. Normalmente esto sería un fallo garrafal: El objeto sería totalmente inútil, porque no estaría conectado al programa de ninguna forma. Pero con el módulo `tkinter`, un objeto `Label` (o cualquier otro elemento) siempre estará conectado al programa mediante su elemento padre. Esto significa que si sabemos que no necesitaremos aceder directamente a un widget, entonces no tenemos que asignarlo a una variable. La principal ventaja de ello es tener un código más corto y limpio.

Hasta ahora hemos sido bastante conservadores, y siempre hemos asignado cada nuevo elemento GUI a una variable. Pero en este caso sabemos que no vamos a acceder a esta etiqueta, por lo que no hemos asignado el objeto `Label` a ninguna variable. En vez de eso, hemos dejado que su elemento padre sea la única referencia a este widget.

A continuación, creamos otra etiqueta exactamente de la misma forma:

```
# create instruction label
Label(self,
      text = "Select all that apply:"
      ).grid(row = 1, column = 0, sticky = W)
```

Esta etiqueta proporciona instrucciones al usuario, ya que le dice que puede elegir tantos tipos de película como quiera.

CREAR CHECK BUTTONS.

Ahora vamos a crear las casillas de verificación, una para cada tipo de película. Vamos a comenzar con la casilla de verificación para las películas de comedia.

Toda casilla de verificación necesita un objeto especial asociado a ella que refleje automáticamente el estado de la casilla. Este objeto especial debe ser una instancia de la clase `BooleanVar` del módulo `tkinter`. Por lo tanto, antes de crear la casilla de verificación para las películas de comedia, debemos instanciar un objeto `BooleanVar` y asignarlo a un nuevo atributo, `likes_comedy`:

```
# create Comedy check button
self.likes_comedy = BooleanVar()
```

A continuación creamos la casilla de verificación en sí misma:

```
Checkbutton(self,
            text = "Comedy",
            variable = self.likes_comedy,
            command = self.update_text
            ).grid(row = 2, column = 0, sticky = W)
```

Este código crea una nueva casilla de verificación con el texto `Comedy`. Al pasar `self.likes_comedy` al parámetro `variable`, lo que hacemos es asociar el estado de la casilla de verificación (seleccionada o no seleccionada) al atributo `likes_comedy`. Al pasar `self.update_text()` al parámetro `command`, asociamos la activación de la casilla de verificación al método `update_text()`. Esto significa que siempre que el usuario seleccione o deseleccione la casilla de verificación, se invocará al método `update_text()`. Finalmente, ubicamos la casilla de verificación en la siguiente línea, alneada a la izquierda.

Notar que que hemos asignado el objeto `Checkbutton` resultante a ninguna variable. Esto es perfectamente legal, porque tod l que nos interesa es su estado, al cual podemos acceder a través del atributo `likes_comedy`.

Después creamos las dos casillas de verificación restantes exactamente de la misma forma:

```
# create Drama check button
self.likes_drama = BooleanVar()
Checkbutton(self,
            text = "Drama",
            variable = self.likes_drama,
            command = self.update_text
            ).grid(row = 3, column = 0, sticky = W)

# create Romance check button
self.likes_romance = BooleanVar()
Checkbutton(self,
            text = "Romance",
            variable = self.likes_romance,
            command = self.update_text
            ).grid(row = 4, column = 0, sticky = W)
```

De esta forma, siempre que el usuario seleccione o deseleccione las casillas Drama o Romance, se invoca al método `update_text()`. Y aunque no asignamos los objetos `Checkbutton` resultantes a ninguna variable, siempre podemos ver el estado de la casilla Drama a través del atributo `likes_drama`, y el estado de la casilla Romance a través del atributo `likes_romance`.

Finalmente, creamos una caja de texto para mostrar las selecciones hechas por el usuario:

```
# create text field to display results
self.results_txt = Text(self, width = 40, height = 5, wrap = WORD)
self.results_txt.grid(row = 5, column = 0, columnspan = 3)
```

OBTENER EL ESTADO DE UN CHECK BUTTON.

Ahora escribimos el método `update_text()`, que actualiza la caja de texto para mostrar las casillas de verificación activadas por el usuario:

```
def update_text(self):
    """ Update text widget and display user's favorite movie types. """
    likes = ""
```

```
        if self.likes_comedy.get():
            likes += "You like comedic movies.\n"

        if self.likes_drama.get():
            likes += "You like dramatic movies.\n"

        if self.likes_romance.get():
            likes += "You like romantic movies."

        self.results_txt.delete(0.0, END)
        self.results_txt.insert(0.0, likes)
```

`tkinter` no permite acceder directamente al valor de un objeto `BooleanVar`. Para ello, debemos llamar la método `get()`. En el código previo, usamos el método `get()` del objeto `BooleanVar` referenciado por `likes_comedy` para obtener el valor del objeto. Si este valor se evalúa a `True`, eso significa que la casilla Comedy está activada, y añadimos la cadena `"You like comedic movies.\n"` a la cadena que estamos construyendo para mostrar en la caja de texto. Efectuamos unas operaciones similares basadas en el estado de las casillas Drama y Romance. Finalmente, borramos todo el texto previo que pudiese haber en la caja de texto, e insertamos la nueva cadena, llamada `likes`, que acabamos de construir.

CONCLUIR EL PROGRAMA.

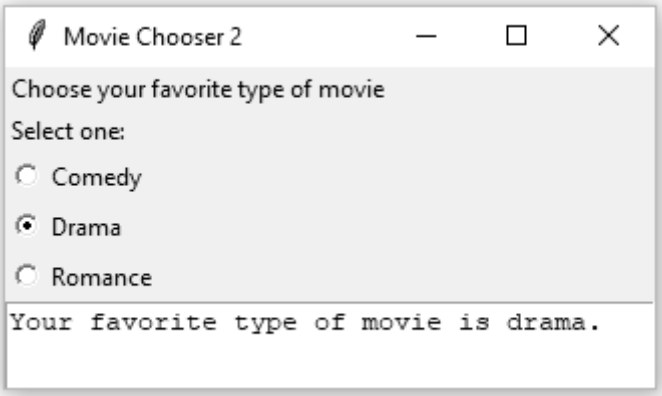
Terminamos el programa con la habitual sección `#main`. Creamos la ventana raíz y un nuevo objeto Application con la ventana raíz como elemento padre. Entonces, arrancamos el bucle de evento de la ventana raíz.

```
# main
root = Tk()
root.title("Movie Chooser")
app = Application(root)
root.mainloop()
```

10.11. USAR RADIO BUTTONS.

Los **botones de opción** (radio buttons) son como las casillas de verificación, excepto que los botones de opción solo permiten seleccionar *uno* de los botones del grupo de botones. Esto viene muy bien si queremos que el usuario haga una única selección de entre un grupo de opcines. Com los botones de opción son tan parecidos a las casillas de verificación, es muy fácil aprender a usarlos.

A modo de ejemplo, vamos a hacer un programa similar al de las películas, solo que ahora usaremos botones de opción en vez de casilla de verificación. El programa le presenta al usuario tres tipos de película diferentes, y el usuario solo puede elegir uno de ellos. La figura muestra la salida del programa:



CONFIGURAR EL PROGRAMA.

Comenzamos el programa importando el módulo `tkinter`:

```
from tkinter import *
```

A continuación escribimos la clase Application. Empezamos definiendo su constructor:

```
class Application(Frame):
    """ GUI Application for favorite movie type. """
    def __init__(self, master):
        """ Initialize Frame. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()
```

Luego creamos etiquetas que le dan instrucciones al usuario:

```
def create_widgets(self):
    """ Create widgets for movie type choices. """
    # create description label
    Label(self,
          text = "Choose your favorite type of movie"
          ).grid(row = 0, column = 0, sticky = W)

    # create instruction label
    Label(self,
          text = "Select one:"
          ).grid(row = 1, column = 0, sticky = W)
```

CREAR RADIO BUTTONS.

Como solo puede estar seleccionado uno de los botones del grupo de botones de acción, no es necesario que cada uno de ellos tenga su propia variable de estado. En vez de eso, todos los botones de acción comparten un único objeto especial que refleja qué botón del grupo está seleccionado. Este objeto puede ser una instancia de la clase `StringVar` del módulo `tkinter`, que permite almacenar y recuperar una cadena. Por lo tanto, antes de crear los propios botones de opción, vamos a crear un único objeto `StringVar` que compartirán todos los botones de opción. Este objeto lo asignaremos al atributo `favorite`, y su valor inicial lo fijaremos a `None` mediante el método `set()`:

```
# create variable for single, favorite type of movie
self.favorite = StringVar()
self.favorite.set(None)
```

A continuación, creamos el botón de opción Comedy:

```
# create Comedy radio button
Radiobutton(self,
            text = "Comedy",
            variable = self.favorite,
            value = "comedy.",
            command = self.update_text
            ).grid(row = 2, column = 0, sticky = W)
```

La opción `variable` de un botón de opción define la variable especial asociada al botón de acción, mientras que la opción `value` define el valor que debe almacenar esa variable especial cuando el botón esté seleccionado. Así pues, fijando la opción `variable` de este botón de acción a `self.favorite` y su opción

value a "comedy.", lo que estamos diciendo es que cuando el botón de opción comedy esté activado, el objeto StringVar referenciado por self.favorite debería almacenar la cadena "comedy.".

Ahora creamos los otros dos botones de opción:

```
# create Drama radio button
Radiobutton(self,
            text = "Drama",
            variable = self.favorite,
            value = "drama.",
            command = self.update_text
            ).grid(row = 3, column = 0, sticky = W)

# create Romance radio button
Radiobutton(self,
            text = "Romance",
            variable = self.favorite,
            value = "romance.",
            command = self.update_text
            ).grid(row = 4, column = 0, sticky = W)
```

Fijando la opción variable del botón de opción Drama a self.favorite y la opción value a "drama.", lo que estamos diciendo es que cuando el botón de opción Drama esté seleccionado, el objeto StringVar reerenciado por self.favorite debería almacenar la cadena "drama.".

Y fijando la opción variable del botón de opción Romance a self.favorite y la opción value a "romance.", lo que estamos diciendo es que cuando el botón de opción Romance esté seleccionado, el objeto StringVar reerenciado por self.favorite debería almacenar la cadena "romance.".

Continuamos creando la caja de texto que muestra el resultado de la selección del usuario:

```
# create text field to display result
self.results_txt = Text(self, width = 40, height = 5, wrap = WORD)
self.results_txt.grid(row = 5, column = 0, columnspan = 3)
```

OBTENER UN VALOR DE UN GRUPO DE BOTONES DE OPCIÓN.

Llamando al método get() del objeto StringVar podemos obtener fácilmente un valor de un grupo de botones de acción:

```
def update_text(self):
    """ Update text area and display user's favorite movie type. """
    message = "Your favorite type of movie is "
    message += self.favorite.get()
```

Construimos la cadena que mostraremos en la caja de texto. Notar que cuando está seleccionado el botón Comedy, sef.favorite.get() devuelve "comedy."; cuando está seleccionado el botón Drama, sef.favorite.get() devuelve "drama."; y cuando está seleccionado el botón Romance, sef.favorite.get() devuelve "romance.".

A continuación, borramos cualquier texto que pudiese estar presente en la caja de texto, y después insertamos la cadena que acabamos de crear, que indica cuál es el tipo de películas favorito del usuario:

```
self.results_txt.delete(0.0, END)
self.results_txt.insert(0.0, message)
```

CONCLUIR EL PROGRAMA.

Terminamos el programa creando la ventana raíz, e instanciando un nuevo objeto `Application`. La GUI se lanza llamando al bucle de evento de la ventana raíz:

```
# main
root = Tk()
root.title("Movie Chooser 2")
app = Application(root)
root.mainloop()
```

10.12. MANEJO DE EVENTOS CON FUNCIONES LAMBDA.

FUNCIONES LAMBDA.

Además de la sentencia `def`, Python también proporciona una forma alternativa para definir funciones: las llamadas **funciones lambda**. Como `def`, esta espresión crea una función a la que se llamará más tarde, pero devuelve directamente la función en vez de asignarla a un nombre de función. Es por esta razón que a las funciones `lambda` se las suele denominar **funciones anónimas**. En la práctica, las funciones `lambda` suelen usarse para definir funciones simples que no necesitan ser referenciadas por su nombre.

La forma general de una expresión lambda consiste en la palabra reservada `lambda`, seguida de uno o más argumentos, seguidos de dos puntos (`:`), seguidos de una expresión que use esos argumentos:

`lambda arg1, arg2, …, argN : expression_using_arguments`
Las funciones definidas por las expresiones lambda funcionan exactamente igual que aquellas definidas por una sentencia `def`, pero hay algunas diferencias que las hacen muy útiles:

- `lambda` es una expresión, no una sentencia. Debido a ello, una expresión lambda puede aparecer en lugares donde una sentencia def no puede, como por ejemplo, dentro de una cadena, o en los argumentos de una llamada a una función. Con la sentencia `def`, las funccciones pueden ser referenciadas por su nombre, pero deben crearse en algún lugar del programa. Pero `lambda` es una expresión y devuelve un valor, que opcionalmente puede asignarse a una variable.

- El cuerpo de una función lambda es una sola expresión, y no un bloque de sentencias. El cuerpo de una función `lambda` es similar a lo que podemos en una sentencia return de una función `def`. Esto es, escribimos el resultado de la función como una expresión pura, en vez de devolverla explícitamente. Y como está limitada a una sola expresión, una función `lambda` es mucho menos general que una función `def`: Las expresiones lambda solo pueden incluir operaciones sencillas que no incluyan sentencias de control, como un `if`. En este sentido, las funciones `lambda` están pensadas para codificar funciones sencillas, mientras que las funciones `def` se usan para codificar tareas más largas y complejas.

A modo de ejemplo, recordemos como definir una función con una sentencia `def`. Escribe lo siguiente en el shell interactivo:

```
>>> def add(x, y, z):
        return x+y+z

>>> add(3, 4, 5)
12
```

Pero también podemos conseguir lo mismo mediante una expresión `lambda`, asignando explícitamente su resultado a una variable, mediante la cual llamaremos después a la función:

```
>>> f = lambda x, y, z: x+y+z
>>> f(3, 4, 5)
12
```

En este código hemos asignado a la variable `f` la función creada por la expresión `lambda`.

¿Para qué podemos usar una función `lambda`? En general, las funciones lambda son bastante útiles como formas de integrar la definición de una función dentro del código que la usa. Esto nos permite no tener que definir una función `def` en aquellos escenarios es lo que solo necesitamos incroporar pequeños trozos de código ejecutable en el lugar donde se necesitan.

Por ejemplo, las expresiones `lambda` se usan de forma muy común en **tablas de saltos**, que no son más que listas o diccionarios con acciones que podemos ejecutar a demanda. Por ejemplo:

```
L = [lambda x: x ** 2,
     lambda x: x ** 3,
     lambda x: x ** 4] # A list of three callable functions

for f in L:
    print(f(2)) # Prints 4, 8, 16

print(L[0](3)) # Prints 9
```

Otro ejemplo sería:

```
from math import *

geometry = {'perimeter': lambda l : 6*l,
            'apothem': lambda l : sqrt (l**2-(l/2)**2),
            'area': lambda p, a: p*a/2}

per = geometry['perimeter'](2)
apo = geometry['apothem'](2)
print("Perimeter: " + str(per) + " meters")
print("Apothem: " + str(apo) + " meters")

area = geometry['area'](per, apo)
print("Area: " + str(area) + " squared meters")
```

FUNCIONES LAMBDA EN TKINTER.

Otro uso muy común de las expresiones lambda es integrar en el propio código dedefinición de un elemento GUI el manejador de evento asociado (siempre que este manejador sea una tarea sencilla). A modo de ejemplo, imaginar que queremos implementar una GUI que simplemente imprima un mensaje en el shell cuando el usuario presione un botón.

Escribe lo siguiente en el shell interactivo, y guarda el programa como `lambda_tkinter(1).py`:

```
from tkinter import *

class Application (Frame):

    def __init__(self, master):
        """ Initialize Frame. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()

    def create_widgets(self):
        label = Label(self, text="Start Page")
        label.grid()
        button = Button(self, text="Print in shell",
                        command=lambda:print("Button pressed!"))
        button.grid()

root = Tk()
root.title ('lambdas GUI demonstrator')
root.geometry("200x100")
app = Application(root)
root.mainloop()
```

Con este programa, cada vez que el usuario presione el botón `Print in shell`, se imprimirá en la ventana del shell la cadena `"Button pressed!"`. Notar que, en este caso, la expresión `lambda` es simplemente una llamada a la función estándar `print`.

La ventaja de usar una expresión `lambda` en vez de `def` es que el código que maneja el evento asociado al botón está justo aquí, en la llamada a la creación del botón. En este caso, la expresión `lambda` permite retrasar la ejecuación del manejador hasta que se produce el evento: La llamada a `print` ocurre cuando el usuario presiona el botón, y no cuando el botón se crea

Las funciones `lambda` también son útiles para solventar un cierto problema que surge con los manejadores de eventos en los programas GUI: Puede que nos hayamos dado cuenta de que la opción `command`, que sirve para asociar un manejador de evento a un cierto elemento GUI, no permite pasar argumentos a las funciones `def` que implementan esos manejadores. En este caso, podemos usar una expresión lambda para hacer una llamada a una función `def` pasándole los argumentos que necesita.

A modo de ejemplo, vamos a impementar el programa previo con una expresión `lambda` que realice una llamada a una función `def`, la cual se encargará de pasar el texto a imprimir por pantalla como argumento. Escribe el siguiente código y guárdalo como `lambda_tkinter(2).py`:

```
from tkinter import *

class Application (Frame):

    def __init__(self, master):
        """ Initialize Frame. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()
```

```
    def create_widgets(self):
        label = Label(self, text="Start Page")
        label.grid()
        button = Button(self, text="Print in shell",
                        command=lambda:self.whenPressed("Hey, I'm passing arguments!"))
        button.grid()

    def whenPressed (self, message):
        print(message)

root = Tk()
root.title ('lambdas GUI demonstrator')
root.geometry("200x100")
app = Application(root)
root.mainloop()
```

En este caso, la función `whenPressed()` simplemente imprime por pantalla la cadena que la expresión lambda le pasa como argumento. Sin embargo, esta técnica es especialmente útil cuando la función a la que llama la expresión `lambda` va a procesar el valor que se le pasa como argumento. A modo de ejemplo, vamos a escribir un programa GUI para una calculadora básica que solamente sume, reste, multiplique, y divida números *enteros*.

10.13. PROYECTO: CALCULADORA.

Vamos a construir el programa GUI para la calculadora paso a paso. Escribe el código en el editor de archivos y guárdalo como `simple_calculator.py`.

IMPORTAR EL MÓDULO TKINTER.

Como siempre, lo primero que debemos hacer es importar el módulo `tkinter`:

```
from tkinter import *
```

EL MÉTODO CONSTRUCTOR DE LA CLASE APPLICATION.

El constructor de la clase `Application` comienza inicializando el objeto `Application` que se haya instanciado, y haciéndolo visible:

```
class Application (Frame):

    def __init__(self, master):
        """ Initialize Frame. """
        super(Application, self).__init__(master)
        self.grid()
```

A continuación definimos e inicializamos la variable global `operationStr`. Esta variable almacenará la cadena que representa la operación a evaluar, y que se mostrará en el display de la calculadora (por ejemplo, `"5*3"`). Esta cadena se irá construyendo progresivamente conforme el usuario vaya tecleando los los dígitos de los operandos y las operaciones a efectuar con ellos. Notar que hemos definido esta variable como global, y que la hemos inicializado a una cadena vacía.

```
""" Define and initialize operationStr global variable. """
global operationStr
self.operationStr=""
```

El método constructor termina invocando al método `create_widgets()` y al método `clear()`, que se encarga de inicializar el display de la calculadora para que muestre la cadena `"0"`.

```
""" Create widgets. """
self.create_widgets()
""" Shows "0" on display when calculator starts. """
self.clear()
```

EL MÉTODO CREATE_WIDGETS() DEL LA CLASE APPLICATION.

Este método crea todos los widgets de la GUI, a saber:

- Una etiqueta sin texto para separar el display de la calculadora del borde superior de la ventana raíz.
- Un elemento de texto de la clase `Entry` para implementar el display de la calculadora. En este elemento de texto es donde mostraremos la cadena de la operación tecleada por el usuario (que se almacena en la variable global `operationStr`), y también el resultado de la operación cuando el usuario presione la tecla `=`.
- Los botones para los dígitos de `0` al `9`, y los botones `+`, `-`, `*`, y `/` para las operaciones de suma, resta, multiplicación, y división.[10]
- El botón `C` para borrar e inicializar a `0` el display de la calculadora.
- El botón `=` para evaluar la operación tecleada en el display.

```
def create_widgets(self):
    self.lbl=Label(self, text="").grid(row=0,column=0,columnspan=4)
    self.display=Entry(self)
    self.display.grid(row=1,column=0,columnspan=4)
    self.button0=Button(self,text=" 0 ",command=lambda:self.btnClick(0)).grid(row=2,column=0)
    self.button1=Button(self,text=" 1 ",command=lambda:self.btnClick(1)).grid(row=2,column=1)
    self.button2=Button(self,text=" 2 ",command=lambda:self.btnClick(2)).grid(row=2,column=2)
    self.button3=Button(self,text=" 3 ",command=lambda:self.btnClick(3)).grid(row=2,column=3)
    self.button4=Button(self,text=" 4 ",command=lambda:self.btnClick(4)).grid(row=3,column=0)
    self.button5=Button(self,text=" 5 ",command=lambda:self.btnClick(5)).grid(row=3,column=1)
    self.button6=Button(self,text=" 6 ",command=lambda:self.btnClick(6)).grid(row=3,column=2)
    self.button7=Button(self,text=" 7 ",command=lambda:self.btnClick(7)).grid(row=3,column=3)
    self.button8=Button(self,text=" 8 ",command=lambda:self.btnClick(8)).grid(row=4,column=0)
    self.button9=Button(self,text=" 9 ",command=lambda:self.btnClick(9)).grid(row=4,column=1)
    self.buttonAdd=Button(self,text=" + ",command=lambda:self.btnClick("+")).grid(row=4,column=2)
    self.buttonSubs=Button(self,text=" - ",command=lambda:self.btnClick("-")).grid(row=4,column=3)
    self.buttonMult=Button(self,text=" * ",command=lambda:self.btnClick("*")).grid(row=5,column=0)
    self.buttonDiv=Button(self,text=" / ",command=lambda:self.btnClick("/")).grid(row=5,column=1)
    self.buttonC=Button(self,text=" C ",command=self.clear).grid(row=5,column=2)
    self.buttonResult=Button(self,text=" = ",command=self.operate).grid(row=5,column=3)
```

La opción `command` para los botones de los dígitos y de los operadores utiliza una expresión `lambda`, que nos permitirá llamar al manejador de evento `btnClick()` pasándole como argumento el valor que representa el botón que el usuario ha presionado. La opción `command` del botón `C` no usa una expresión `lambda`, porque simplemente llama (sin pasar argumentos) a la función `clear()` para limpiar el display de la calculadora. De forma similar, la opción `command` del botón `=` llama directamente a la función `operate()`, que se encarga de evaluar la cadena de la operación tecleada por el usuario.

EL MÉTODO CLEAR() DE LA CLASE APPLICATION.

Este método permite borrar el display de la calculadora al arrancar el programa, y cuando el usuario presiona el botón `C`. Como vemos, esta función simplemente asigna a la variable global `operationStr` una cadena vacía, invoca al método `delete()` del elemento de texto `display` para borrar su contenido actual, y llama al método `insert()` de `display` para insertar la cadena `"0"`.

[10] Por cierto, que el botón de resta también servirá para añadir un signo negativo delante de un número.

```
def clear(self):
    """ Clears and shows "0" on the display
        of the calculator. """
    self.operationStr=""
    self.display.delete(0, END)
    self.display.insert(0, "0")
```

Notar que, como `display` es un elemento tipo `Entry` (esto es, una sola línea de texto), el punto de inicio que debemos proporcionarle para borrar e insertar texto dentro de él solo necesita una coordenada, en lugar de las dos coordenadas que necesita un elemento de texto tipo `Text`.

EL MÉTODO BTNCLICK() DE LA CLASE APPLICATION.

Este método se encarga de construir la cadena de la operación matemática a evaluar conforme el usuario va presionando los botones de los dígitos y de los operadores.

```
def btnClick(self, num):
    """ Builds the operation string
        and shows it on the display. """
    self.operationStr=self.operationStr+str(num)
    self.display.delete(0, END)
    self.display.insert(0, self.operationStr)
```

La función recibe como argumento el valor pasado por la correspondiente expresión `lambda` del manejador de evento del botón que se haya presionado (a saber, el número entero `4` si se presionó el botón del `4`, la cadena `"-"` si se presionó el botón de resta, etc.). Esta función simplemente concatena el valor recibido con el contenido actual de la cadena `operationStr`, donde se guarda la cadena de la operación. A continuación, el método borra el contenido actual del display, e inserta el nuevo valor de la cadena `operationStr`. Por ejemplo, imaginar que el usuario presiona los botones del `4`, de la resta, y del `2`. A cada pulsación, el método concatena los valores `4`, `"-"`, y `2` en la variable `operationStr`, y los muestra progresivamente en el display: Al pulsar el botón del `4`, el método almacena el valor `"4"` en `operationStr`, y a continuación, lo muestra en el display. Al pulsar el botón de la resta, `operationStr` toma el valor `"4-"`, que pasa a mostrarse en el display. Finalmente, al pulsar el botón del `2`, `operationStr` toma el valor `"4-2"`, que después se inserta en el display.

LA FUNCIÓN EVAL.

Antes de escribir el método `operate()` de la clase `Application`, vamos a hacer una pausa para explicar la utilidad de la función `eval()`.

`eval()` es una función integrada de Python, lo que significa que no hace falta importar ningún módulo para poder usarla. Esta función evalúa una expresión matemática que le pasamos como argumento en forma de cadena, devolviendo como resultado un valor numérico.

Escribe lo siguiente en el shell interactivo:

```
>>> eval('5+2')
7
>>> eval('(5+2)*3')
21
>>> print('The avarage of 2, 6, and 7 is: ' + str(eval('(2+6+7)/3')))
The avarage of 2, 6, and 7 is: 5.0
```

Como veremos a continuación, la función `eval()` es casi todo lo que necesitamos para evaluar la expresión matemática tecleada por el usuario en el display de la calculadora.

EL MÉTODO OPERATE() DE LA CLASE APPLICATION.

Este método se invoca cuando el usuario presiona la tecla = de la calculadora. La tarea de este método es evaluar la operación matemática introducida por el usuario:

```
def operate(self):
    """ Performs the operation previously built
        and shown on the display. """
    try:
        op=str(eval(self.display.get()))
    except:
        self.clear()
        op=("ERROR")

    """ Shows the result of the operation
        on the display. """
    self.display.delete(0, END)
    self.display.insert(0, op)
```

Para ello, el método obtiene el contenido del elemento de texto `display`, que es simplemente la cadena que representa la operación tecleada por el usuario. A continuación, la función `eval()` evalúa esa expresión matemática. Finalmente, la función `str()` convierte el resultado en una cadena, que se almacena en la variable `op`. (Notar que si esta instrucción falla, el manejador de excepciones simplemente borra el display y almacena en `op` un mensaje de error). A continuación, el método borra el contenido actual del display (que será la operación que ha tecleado el usuario previamente), e inserta en su lugar el contenido de la variable `op` (que es el resultado de la operación, o un mensaje de error).

EL PROGRAMA PRINCIPAL.

Como siempre, creamos la ventana raíz e instanciamos un objeto `Application`. Finalmente, arrancamos la GUI llamando al método `mainloop()` de la ventana raíz.

```
# main
root=Tk()
root.title("CALCULATOR")
root.geometry("120x160")
app = Application(root)
root.mainloop()
```

10.14. PROYECTO: MAD LIB.

Ahora que ya sabemos cómo manejar una variedad de elementos GUI de forma individual, vams a combinarlos todos en una única GUI que implemente el programa Mad Lib del que hablamos en la sección 10.2. Aquí no vamos a introducir nuevos conceptos, asñi que las explicaciones del código no serán muy extensas. Escribe el siguente código en el editor de archivos y guárdalo como `mad_lib.py`:

```
from tkinter import *

class Application(Frame):
    """ GUI application that creates a story based on user input. """
    def __init__(self, master):
        """ Initialize Frame. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()
```

```
def create_widgets(self):
    """ Create widgets to get story information and to display story. """
    # create instruction label
    Label(self,
          text = "Enter information for a new story"
          ).grid(row = 0, column = 0, columnspan = 2, sticky = W)

    # create a label and text entry for the name of a person
    Label(self,
          text = "Person: "
          ).grid(row = 1, column = 0, sticky = W)
    self.person_ent = Entry(self)
    self.person_ent.grid(row = 1, column = 1, sticky = W)

    # create a label and text entry for a plural noun
    Label(self,
          text = "Plural Noun:"
          ).grid(row = 2, column = 0, sticky = W)
    self.noun_ent = Entry(self)
    self.noun_ent.grid(row = 2, column = 1, sticky = W)

    # create a label and text entry for a verb
    Label(self,
          text = "Verb:"
          ).grid(row = 3, column = 0, sticky = W)
    self.verb_ent = Entry(self)
    self.verb_ent.grid(row = 3, column = 1, sticky = W)

    # create a label for adjectives check buttons
    Label(self,
          text = "Adjective(s):"
          ).grid(row = 4, column = 0, sticky = W)

    # create itchy check button
    self.is_itchy = BooleanVar()
    Checkbutton(self,
                text = "itchy",
                variable = self.is_itchy
                ).grid(row = 4, column = 1, sticky = W)

    # create joyous check button
    self.is_joyous = BooleanVar()
    Checkbutton(self,
                text = "joyous",
                variable = self.is_joyous
                ).grid(row = 4, column = 2, sticky = W)

    # create electric check button
    self.is_electric = BooleanVar()
    Checkbutton(self,
                text = "electric",
                variable = self.is_electric
                ).grid(row = 4, column = 3, sticky = W)

    # create a label for body parts radio buttons
    Label(self,
          text = "Body Part:"
          ).grid(row = 5, column = 0, sticky = W)

    # create variable for single, body part
    self.body_part = StringVar()
    self.body_part.set(None)
```

```
        # create body part radio buttons
        body_parts = ["bellybutton", "big toe", "medulla oblongata"]
        column = 1
        for part in body_parts:
            Radiobutton(self,
                        text = part,
                        variable = self.body_part,
                        value = part
                        ).grid(row = 5, column = column, sticky = W)
            column += 1

        # create a submit button
        Button(self,
               text = "Click for story",
               command = self.tell_story
               ).grid(row = 6, column = 0, sticky = W)

        self.story_txt = Text(self, width = 75, height = 10, wrap = WORD)
        self.story_txt.grid(row = 7, column = 0, columnspan = 4)

    def tell_story(self):
        """ Fill text box with new story based on user input. """
        # get values from the GUI
        person = self.person_ent.get()
        noun = self.noun_ent.get()
        verb = self.verb_ent.get()
        adjectives = ""
        if self.is_itchy.get():
            adjectives += "itchy, "
        if self.is_joyous.get():
            adjectives += "joyous, "
        if self.is_electric.get():
            adjectives += "electric, "
        body_part = self.body_part.get()

        # create the story
        story = "The famous explorer "
        story += person
        story += " had nearly given up a life-long quest to find The Lost City of "
        story += noun.title()
        story += " when one day, the "
        story += noun
        story += " found "
        story += person + ". "
        story += "A strong, "
        story += adjectives
        story += "peculiar feeling overwhelmed the explorer. "
        story += "After all this time, the quest was finally over. A tear came to "
        story += person + "'s "
        story += body_part + ". "
        story += "And then, the "
        story += noun
        story += " promptly devoured "
        story += person + ". "
        story += "The moral of the story? Be careful what you "
        story += verb
        story += " for."

        # display the story
        self.story_txt.delete(0.0, END)
        self.story_txt.insert(0.0, story)
```

```
# main
root = Tk()
root.title("Mad Lib")
app = Application(root)
root.mainloop()
```

IMPORTAR EL MÓDULO TKINTER.

Como ya sabemos, lo primero que debemos hacer es importar el módulo `tkinter`:

```
from tkinter import *
```

EL MÉTODO CONSTRUCTOR DE LA CLASE APPLICATION.

Como todos los métodos constructores de la clase `Application` que hemos escrito antes, éste inicializa el objeto `Application` recién creado e invoca al método `create_widgets()`:

```
class Application(Frame):
    """ GUI application that creates a story based on user input. """
    def __init__(self, master):
        """ Initialize Frame. """
        super(Application, self).__init__(master)
        self.grid()
        self.create_widgets()
```

EL MÉTODO CREATE_WIDGETS() DE LA CLASE APPLICATION.

Este método crea todos los widgets de la GUI. Lo único nuevo que hacemos es crear los tres botones de opción mediante un bucle que itera a través de una lista de cadenas, y que fija los valores de las opciones `text` y `value` de cada botón:

```
def create_widgets(self):
    """ Create widgets to get story information and to display story. """
    # create instruction label
    Label(self,
          text = "Enter information for a new story"
          ).grid(row = 0, column = 0, columnspan = 2, sticky = W)

    # create a label and text entry for the name of a person
    Label(self,
          text = "Person: "
          ).grid(row = 1, column = 0, sticky = W)
    self.person_ent = Entry(self)
    self.person_ent.grid(row = 1, column = 1, sticky = W)

    # create a label and text entry for a plural noun
    Label(self,
          text = "Plural Noun:"
          ).grid(row = 2, column = 0, sticky = W)
    self.noun_ent = Entry(self)
    self.noun_ent.grid(row = 2, column = 1, sticky = W)

    # create a label and text entry for a verb
    Label(self,
          text = "Verb:"
          ).grid(row = 3, column = 0, sticky = W)
    self.verb_ent = Entry(self)
    self.verb_ent.grid(row = 3, column = 1, sticky = W)
```

```
# create a label for adjectives check buttons
Label(self,
      text = "Adjective(s):"
      ).grid(row = 4, column = 0, sticky = W)

# create itchy check button
self.is_itchy = BooleanVar()
Checkbutton(self,
            text = "itchy",
            variable = self.is_itchy
            ).grid(row = 4, column = 1, sticky = W)

# create joyous check button
self.is_joyous = BooleanVar()
Checkbutton(self,
            text = "joyous",
            variable = self.is_joyous
            ).grid(row = 4, column = 2, sticky = W)

# create electric check button
self.is_electric = BooleanVar()
Checkbutton(self,
            text = "electric",
            variable = self.is_electric
            ).grid(row = 4, column = 3, sticky = W)

# create a label for body parts radio buttons
Label(self,
      text = "Body Part:"
      ).grid(row = 5, column = 0, sticky = W)

# create variable for single, body part
self.body_part = StringVar()
self.body_part.set(None)

# create body part radio buttons
body_parts = ["bellybutton", "big toe", "medulla oblongata"]
column = 1
for part in body_parts:
    Radiobutton(self,
                text = part,
                variable = self.body_part,
                value = part
                ).grid(row = 5, column = column, sticky = W)
    column += 1

# create a submit button
Button(self,
       text = "Click for story",
       command = self.tell_story
       ).grid(row = 6, column = 0, sticky = W)

self.story_txt = Text(self, width = 75, height = 10, wrap = WORD)
self.story_txt.grid(row = 7, column = 0, columnspan = 4)
```

EL MÉTODO TELL_STORY() DE LA CLASE APPLICATION.

En este método obtenemos los valores que ha introducido el usuario y los usa para crear una única gran cadena para la historia. Después, borramos cualquier texto previo que pudiese haber en la caja de texto, e insertamos la nueva cadena para mostrar al usuario a historia que ha creado.

```
def tell_story(self):
    """ Fill text box with new story based on user input. """
    # get values from the GUI
    person = self.person_ent.get()
    noun = self.noun_ent.get()
    verb = self.verb_ent.get()
    adjectives = ""
    if self.is_itchy.get():
        adjectives += "itchy, "
    if self.is_joyous.get():
        adjectives += "joyous, "
    if self.is_electric.get():
        adjectives += "electric, "
    body_part = self.body_part.get()

    # create the story
    story = "The famous explorer "
    story += person
    story += " had nearly given up a life-long quest " + \
             "to find The Lost City of "
    story += noun.title()
    story += " when one day, the "
    story += noun
    story += " found "
    story += person + ". "
    story += "A strong, "
    story += adjectives
    story += "peculiar feeling overwhelmed the explorer. "
    story += "After all this time, the quest was finally over. " + \
             "A tear came to "
    story += person + "'s "
    story += body_part + ". "
    story += "And then, the "
    story += noun
    story += " promptly devoured "
    story += person + ". "
    story += "The moral of the story? Be careful what you "
    story += verb
    story += " for."

    # display the story
    self.story_txt.delete(0.0, END)
    self.story_txt.insert(0.0, story)
```

EL PROGRAMA PRINCIPAL.

Ya hemos visto esta parte del código muchas veces. Creamos la ventana raíz e instanciamos un objeto `Application`. Finalmente, arrancamos la GUI llamando al método `mainloop()` de la ventana raíz.

```
# main
root = Tk()
root.title("Mad Lib")
app = Application(root)
root.mainloop()
```

10.15. EJERCICIOS DEL CAPÍTULO 10.

Ejercicio 10.1. Escribe un programa GUI con un botón `saludo`. Al pinchar el botón, el programa imprime en el shell un saludo genérico. Añade las etiquetas que creas necesarias para explicar al usuario el funcionaiento del programa. Guarda el programa como `Ejer10.1.py`.

Ejercicio 10.2. Escribe un programa GUI con dos elementos de texto (`nombre` y `apellido`) y un botón `mostrar`. El usuario inserta su nombre y su primer apellido en los respectivos elementos de texto, y cuando el usuario presiona el botón Mostrar, el programa escribe en el shell interactivo una salida como la siguiente:

```
First Name: Alejandro
Last Name: Martínez
```

Añade las etiquetas que creas necesarias para explicar al usuario el funcionamiento del programa. Guarda el programa como `Ejer10.2.py`.

Ejercicio 10.3. Escribe un programa GUI en el que haya un elemento de texto y dos botones, llamados `saludar` y `borrar`. Cuando el usuario clica en el botón `saludar`, el programa escribe un saludo en el elemento de texto. Cuando el usuario clica en `borrar`, el programa borra el texto presente en el elemento de texto. Añade las etiquetas que creas necesarias para explicar al usuario el funcionamiento del programa. Guarda el programa como `Ejer10.3.py`.

Ejercicio 10.4. Escribe un programa GUI en el que haya dos elemento de texto y un botón, llamado `saludar`. En uno de los elemento de texto, el usuario escribe su nombre. Al pinchar en el botón, el programa muestra en el segundo elemento de texto un saludo personalizado para el usuario. Añade las etiquetas que creas necesarias para explicar al usuario el funcionaiento del programa. Guarda el programa como `Ejer10.4.py`.

Ejercicio 10.5. Escribe un programa GUI con dos botones, `red` y `blue`. El programa comienza con la ventana raíz con su color por defecto. Cuando el usuario clica en el botón `red` o en el botón `blue`, el programa cambia el color de fondo de la ventana al color correspondiente.

PISTA: Para cambiar el color de fondo de cualquier elemento GUI, hay que configurar su opción `background`.

```
root = Tk()
root.configure(bg = 'black')
```

Guarda el programa como `Ejer10.5.py`.

Ejercicio 10.6. Escribe un programa GUI en el que haya tres elementos de texto (`nombreUsuario`, `contraseña`, y `comunicaciones`) y un botón `enviar`. El usuario introduce un nombre de usuario y una contraseña, que se compara contra un diccionario de usuarios y contraseñas válidos. Si el usuario introduce un nombre y una contraseña válidos, el programa muestra (en el elemento de texto `comunicaciones`) el

mensaje `Acceso concedido`. En caso contrario, muestra el mensaje `Acceso denegado`. Guarda el programa como `Ejer10.6.py`.

Ejercicio 10.7. Escribe un programa GUI en el que haya una etiqueta y un botón llamado `iniciar`. Al presionar el botón, la etiqueta comienza a contar segundos de uno en uno.

PISTA: Necesitarás el método `after()`, que está disponible para cualquier elemento GUI, y que permite llamar a una función después de que haya transcurrido un cierto tiempo expresado en ms:

```
element_name.after(time_in_ms, function_name)
```

Guarda el programa como `Ejer10.7.py`.

Ejercicio 10.8. Agenda.
Escribe un programa GUI que almacene los números de teléfono y las direcciones de correo de un grupo de personas. (Guarda esta información en una estructura de datos tipo diccionario). La GUI tiene un botón para cada persona en la agenda, y al clicar sobre uno de ellos, muestra su información de contacto como muestra la figura. Guarda el programa como `Ejer10.8.py`.

Ejercicio 10.9. Puerta secreta.
Escribe un programa GUI en el que haya tres botones llamados `puerta1`, `puerta2`, y `puerta3`, y un elemento de texto llamado `resultado`. Una etiqueta le pregunta al usuario detrás de qué puerta está escondido el premio. El programa decide aleatoriamente tras qué puerta (1, 2, ó 3) se esconde el premio. El usuario pincha en el botón de la puerta tras la cual cree que se encuentra el premio. Si el usuario acierta, el programa muestra una felicitación en el elemento de texto. Si el usuario falla, el programa le dice que ha errado, y le indica tras qué puerta estaba escondido el premio. Guarda el programa como `Ejer10.9.py`.

Ejercicio 10.10. Escribe una versión del programa de adivinar el número (sección 3.9) usando una GUI. Tienes total libertad para diseñar la GUI a tu gusto. Guarda el programa como `Ejer10.10.py`.

Ejercicio 10.11. Crea un programa GUI, llamado Order up!, que presente al usuario un (sencillo) menú de restaurante, que liste los platos y los precios. El usuario puede elegir diferentes platos, y entonces el programa muestra la cuenta a pagar. Tienes total libertad para diseñar la GUI a tu gusto. Guarda el programa como `Ejer10.11.py`.

Ejercicio 10.12. Calculadora básica (V.2).
Escribe una GUI que implemente una calculadora básica similar a la que programamos en la sección 10.13, pero con unos requisitos de funcionamiento algo distintos:

- El usuario escribe mediante el teclado del ordenador dos números a operar en sendos elementos de texto.

- La GUI dispone de 4 botones para seleccionar la operación que se efectuará con esos dos números, a saber, suma, resta, multiplicación, y división.
- Tras haber introducido los dos números, y al presionar en uno de los botones de operación, el programa muestra el resultado de esa operación en un tercer elemento de texto.
- En este programa está prohibido el uso de la función `eval()`.

Guarda el programa como `Ejer10.12.py`.

Ejercicio 10.13. Calculadora de préstamos.
La siguiente fórmula permite calcular préstamo todavía por saldar (o balance actual) B_k después de k pagos mensuales de una cantidad p de dinero, comenzando con un saldo inicial (préstamo inicial, "loan principal") B_0 y un interés peródico anual r ("period rate"):

$$B_k = (1+r)^k B_0 - \frac{(1+r)^k - 1}{r} p$$

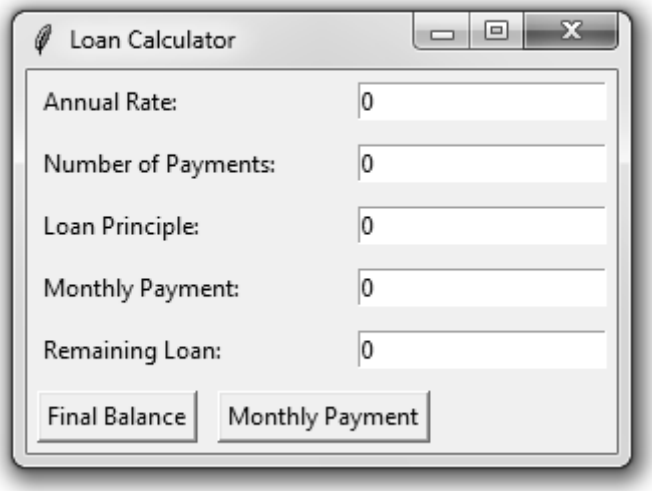
, donde el interés anual (int) se expresa en un porcentaje (por ejemplo, el 3%), $i = int/100$ es el interés anual en forma decimal, $r = i/12$ es el interés peródico anual, B_0 es el préstamo inicial, B_k es el préstamo por saldar después de k pagos, y p es el pago mensual.

Si queremos hallar el pago mensual necesario para que el préstamo quede saldado en n pagos, la fórmula es:

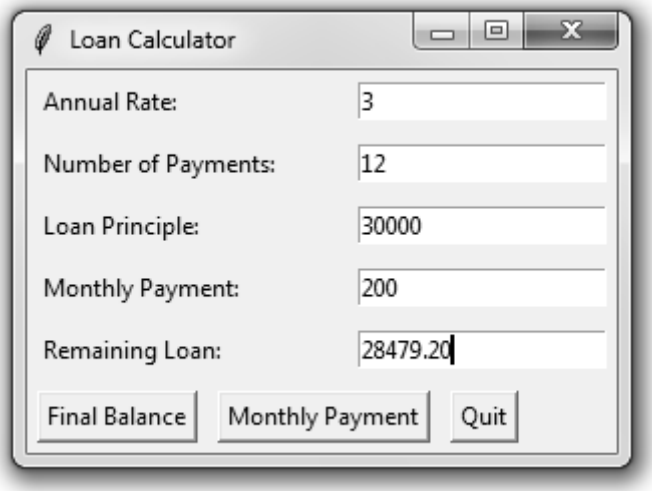
$$p = B_0 \frac{r(1+r)^n}{(1+r)^n - 1}$$

Escribe un aprograma GUI en el que el usuario *pueda* insertar (en varios elementos de texto) el interés anual (int), el número de pagos (número de pagos ya realizados (k), o el número total de pagos deseado (n)), el préstamos inicial (B_0), el pago mensual (p), y el préstamo por saldar (B_k). El programa dispone de dos botones, Saldo Final y Pago Mensual.

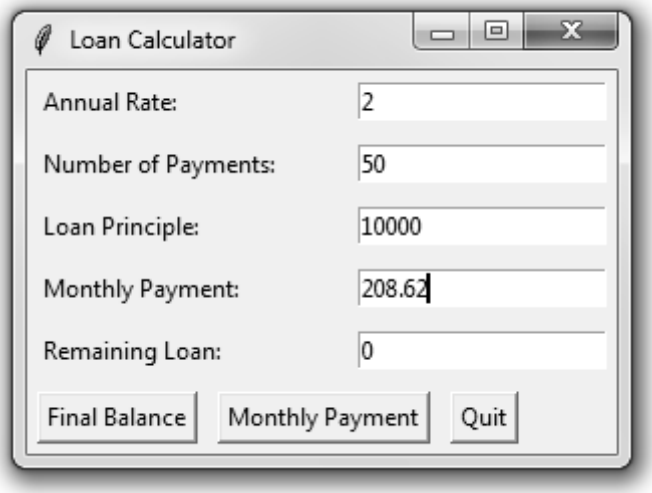
El programa debe funcionar como sigue:
a) Si el usuario inserta el interés anual int, el número de pagos ya realizados k, el préstamo inicial B_0, y el pago mensual actual p, al clicar el botón Saldo Final, el programa debe calcular (y mostrar en el elemento de texto para el préstamo por saldar) la cantidad de préstamo que le queda por pagar, B_k, tras k pagos.
b) Si el usuario inserta el interés anual int, el número total de pagos deseado n, y el préstamo inicial B_0, al clicar en el botón Pago Mensual, el programa debe calcular (y mostrar en el elemento de texto para el pago mensual) cuál debería ser su pago mesual para saldar el préstamo en n pagos.



Ventana inicial.



Opción a: Cálculo del préstamo por saldar tras k pagos.



Opción b: Cálculo del pago mensua para saldar un préstamo en n pagos.

Guarda el programa como `Ejer10.13.py`.

Ejercicio 10.14. Calculadora científica.
En la sección 10.13 programamos una calculadora básica que únicamente nos permitía sumar, restar, multiplicar, y dividir números enteros. En este ejercicio vamos a mejorar esa calculadora añadiéndole las siguentes funcionalidades:

- Debemos añadir un botón para permitir que el usuario pueda insertar núemros decimales.
- Debemos ampliar el número de operaciones que permite efectuar la calculadora, añadiendo botones para realizar las operaciones de exponenciación, raíz cuadrada, logaritmo natural, e^x, logaritmo en base 10, y 10^x.
- Además, debemos añadir un botón que nos permita obtener un número flotante aleatorio etre 0 y 1, que se añade a la cadena de la expresión matemática a evaluar.

Guarda el programa como `Ejer10.14.py`.

11. DEPURACIÓN DE ERRORES (DEBUGGING).

Ahora que ya sabemos lo suficiente como para escribir programas Python relativamente complejos, es probable que en ellos generemos errores y malfuncionamientos (bugs) que no son fáciles de encontrar. En este capítulo aprenderemos a usar algunas herramientas y técnicas para detectar la causa raíz de los malfuncionamientos de nuestros programas, para conseguir corregirlos de forma más rápida y con menos esfuerzo.

11.1. INTRODUCCIÓN.

Un ordenador simplemente ejecutará ciegamente aquello que le digamos que haga; no podemos esperar que una máquina sea capaz de interpretar un contexto y adivinar cuáles son nuestras intenciones al escribir un programa. Incluso los programadores profesionales constantemente generan malfuncionamientos en sus programas, por lo que no debemos desanimarnos si nuestro programa no funciona como esperábamos, o no funciona en absoluto.

Afortunadamente, hay unas cuantas técnicas y herramientas que permiten visualizar qué es lo que está haciendo nuestro código, e identificar qué está funcionando mal. En primer lugar, echaremos un vistazo a los **registros** ("loggings") y a las **afirmaciones** ("assertions"), dos características que nos pueden ayudar a detectar malfuncionamientos rápidamente. En general, cuanto antes detectemos un malfuncionamiento, más fácil será de corregir.

Después, aprenderemos a usar el **depurador de errores** ("debugger"). El **debugger** es una herramienta del IDLE que ejecuta un programa instrucción a instrucción, y que permite inspeccionar los valores almacenados en las variables conforme se va ejecutando el código para rastrear la forma en la que van cambiando durante la ejecución del programa. Este proceso es mucho más lento que ejecutar el programa sin más, pero es muy útil para ver los valores que van tomando las variables, en vez de deducir del código fuente qué valores deberían tomar.

11.2. LANZAR EXCEPCIONES.

Python lanza una excepción siempre que intenta ejecutar un código inválido. En el capítulo 4 aprendimos cómo manejar excepciones con las sentencias `try` y `except`, para que nuestro programa pudiese salvar ciertas excepciones por anticipado. Pero nosotros también podemos lanzar nuestras propieas excepciones en nuestro código. Lanzar una excepción es una forma de decir: "Deja de ejecutar el código de esta función y lleva el flujo de ejecución del programa a la sentencia `except`".

Las excepcines se lanzan usando una sentencia `raise`. Escrita en Python, una sentencia `raise` consta de lo siguiente:

- La palabra reservada `raise`.
- Una llamada a la fución `Exception()`.
- Una cadena con un mensaje de error útil para pasárselo a la función `Exception()`.

A modo de ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> raise Exception('This is the error message')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    raise Exception('This is the error message')
Exception: This is the error message
```

Si no hay unas sentencias `try` y `except` que cubran la sentencia `raise` que lanzó la excepción, el programa simplemente falla y muestra el mensaje de error asociado a la excepción.

A menudo es el código que llama a la función, y no la función en sí misma, el que sabe cómo manejar una excepción. Por lo tanto, es muy común ver una sentencia `raise` dentro de una función, y ver las sentencias `try` y `except` en el código que llama a la función. Abre una nueva ventana del editor de achivos, escribe el siguiente código, y guárdalo como `boxPrint.py`:

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
❶       raise Exception('Symbol must be a single character string.')
    if width <= 2:
❷       raise Exception('Width must be greater than 2.')
    if height <= 2:
❸       raise Exception('Height must be greater than 2.')

    print(symbol * width)
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)

for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        boxPrint(sym, w, h)
❹   except Exception as err:
❺       print('An exception happened: ' + str(err))
```

Aquí hemos definido una función `boxPrint()` que recibe un carácter, una anchura, y una altura, y usa el carácter recibido para dibujar por pantalla una caja con esa anchura y altura.

Imaginar que queremos que el carácter sea un único carácter, y que la anchura y la altura sean mayores que 2. Añadimos unas sentencias `if` para lanzar excepciones si estos requerimientos no se satisfacen. Después, cuando llamamos a la función `boxPrint()` pasándole distintos argumentos, nuestras sentencias `try/except` se encargarán de manejar los argumentos que no sean válidos.

Este programa usa la forma `except Exception as err` de la sentencia `except` (4), que funciona como indicamos a continuación: Si en las líneas (1), (2), ó (3) en `boxPrint()` devuelven un objeto `Exception`, la sentencia `except` en (4) almacenará este objeto e una variable llamada `err`. Después, este objeto `Exception` puede convertirse en una cadena mediante la función `str()` para producir un mensaje de error para el usuario (5). Al ejecutar este programa, la salida debería parecerse a la siguiente:

```
****
*  *
*  *
****
OOOOOOOOOOOOOOOOOOOO
O                  O
O                  O
O                  O
OOOOOOOOOOOOOOOOOOOO
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.
```

Las sentencias `try` y `except` nos permiten capturar errores para impedir que el programa simplemente falle y se detenga.

11.3. OBTENER UN RASTREO COMO UNA CADENA.

Cuendo Python se encuentra con un error produce una gran cantidad de información oculta acerca de ese error, llamada **rastreo** ("traceback"). El rastreo incluye el mensaje de error, el número de la línea de código que causó el error, y la secuancia de llamadas a funciones que condujo a ese error. A esta secuancia de llamadas se la denomina **pila de llamadas**.

Abre una nueva ventana del editor de archivos, escribe este código, y guárdalo como `errorExample.py`:

```
def spam():
    bacon()

def bacon():
    raise Exception('This is the error message')

spam()
```

Al ejecutar este programa, la salida debería parecerse a lo siguiente:

```
Traceback (most recent call last):
  File "C:/ errorExample.py", line 7, in <module>
    spam()
  File "C:/ errorExample.py", line 2, in spam
    bacon()
  File "C:/ errorExample.py", line 5, in bacon
    raise Exception('This is the error message')
Exception: This is the error message
```

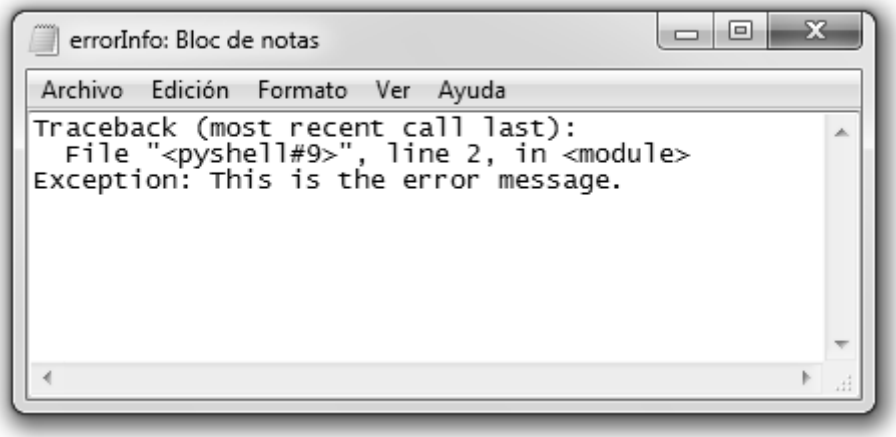
De este rastreo, vemos que el error ocurrió en la línea 5, en la función `bacon()`. Esta llamada en particular a `bacon()` provino de la línea 2, en la función `spam()`, que a su vez fue llamada en la linea 7. En programas donde se puede llamar a las funciones desde distintos sitios, la pila de llamadas puede ayudarnos a determinar qué llamada produjo el error.

Python muestra automáticamente un rastreo siempre que ocurre una excepción que no se captura. Pero también podemos obtener este rastero en forma de cadena, llamando a la función `traceback.format_exc()`. Esta función es útil si queremos guardar la información relativa al rastreo de una excepción, pero también cuendo queremos hacer que una sentencia `except` capture esa excepción. Para poder llamar a esta función, debemos importar el módulo `traceback`.

Así, en vez de dejar que un programa simplemente falle y se detenga cuando ocurre una excepción, podemos escribir la información de rastero en un archivo de registro, y hacer que el programa siga ejecutándose. Mástarde podemos acudor al archivo de registro, cuando estemos listos para depurar el programa. Escribe lo siguiente en el shell interactivo:

```
>>> import traceback
>>> try:
        raise Exception('This is the error message.')
except:
        errorFile = open('errorInfo.txt', 'w')
        errorFile.write(traceback.format_exc())
        errorFile.close()
        print('The traceback info was written to errorInfo.txt.')


115
The traceback info was written to errorInfo.txt.
```

El número 115 es el valor de retorno del método `write()`, ya que se escribieron 115 caracteres en el archivo. El texto del rastreo se ha escrito en el archivo *errorInfo.txt*.



11.4. AFIRMACIONES (ASSERTIONS).

Una **afirmación** (assertion) es una comprobación para asegurarnos de que nuestro código no está haciendo algo que es evidentemente incorrecto. Estas comprobaciones se realizan mediante sentencias `assert`. Si esta comprobación falla, se lanza una excepción de tipo `AssertionError`. En Python, una sentencia `assert` consta de lo siguiente:

- La palabra reservada `assert`.
- Una condición (esto es, una expresión que se evalúa a `True` o `False`).
- Una coma.
- La cadena que se mostrará cuando la condición se evelúe a `False`.

A modo de ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> podBayDoorStatus = 'open'
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
>>> podBayDoorStatus = "I'm sorry, Dave. I'm afraid I can't do that."
>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

Aquí hemos comenzado fijando el valor de `podBayDoorStatus` (estado de la puerta de un compartimento) a `'open'`, por lo que a partir de esta línea esperamos que el valor de esta variable sea `'open'`. En un programa que use esta variable podríamos haber escrito un montón de código que asuma que el valor de la variable es `'open'` (esto es, podríamos haber escrito un código que solo funcione correctamente si esta variable almacena el valor `'open'`). Por esa razón hemos añadido una afirmación para asegurarnos de que, en efecto, el valor de `podBayDoorStatus` es `'open'`. En esta sentencia incluimos el mensaje `'The pod bay doors need to be "open".'` para poder ver fácilmente qué es lo que está fallando cuando esta suposición no se cumple.

Imaginar que después cometemos el error de asignar a `podBayDoorStatus` otro valor distinto, y suponer que no nos damos cuenta de ello. La sentencia de afirmación captura este error y lanza una excepción que nos informa de qué ha ido mal.

En otras palabras, una sentencia `assert` dice lo siguiente: "Afirmo de que esta condición debe ser cierta, y si no es así, hay un error en alguna parte del programa". Al contrario que hacemos con las excepciones,

nuestros programas nunca deberían tratar de manejar afirmacines con una sentencias `try – except`; si una afirmación falla, el programa debería terminar. Esto nos permitirá detectar con más facilidad el origen del error, reduciendo drásticamente la cantidad de código a comprobar para encontrar ese error.

Cuidado: Las afirmaciones solo sirven para detectar errores de programación, no errores de usuario. Para manejar errores de los que el programa se puede recuperar (como no poder encontrar un archivo, o datos inválidos insertados por el usuario) debemos lanzar una excepción, y no detectarlos con una afirmación.

USAR UNA AFIRMACIÓN EN LA SIMULACIÓN DE UN SEMÁFORO.

Imaginar que estamos construyendo un simulador para un semáforo. La estructura de datos que representa las luces en una intersección es un diccionario con las claves `'ns'` y `'ew'`, para la luz que mira en la dirección norte – sur, y para la luz que mira en la dirección este – oeste, respectivamente. Los valores posibles para estas claves serán las cadenas `'green'`, `'yellow'`, o `'red'`. El código se parecería a lo siguiente:

```
market_2nd = {'ns': 'green', 'ew': 'red'}
mission_16th = {'ns': 'red', 'ew': 'green'}
```

Estas dos variables representarán los semáforos para los cruces de las calles Market Street y 2nd Street, y de Mission Street con 16th Street. Para comenzar con el proyecto, vamos a escribir una función `switchLights()`, que recibirá el diccionario que representa a un semáforo como argumento y cambiará sus luces.

A primera vista puede que pensemos que `switchLights()` solo debería cambiar cada luz al siguiente color de la secuencia (esto es, todo valor a `'green'` debería cambiarse a `'yellow'`, todo valor a `'yellow'` debería cambiarse a `'red'`, y todo valor a `'red'` debería cambiarse a `'green'`). El código que implementa esta idea se parecería a lo siguiente:

```
def switchLights(stoplight):
    for key in stoplight.keys():
        if stoplight[key] == 'green':
            stoplight[key] = 'yellow'
        elif stoplight[key] == 'yellow':
            stoplight[key] = 'red'
        elif stoplight[key] == 'red':
            stoplight[key] = 'green'

switchLights(market_2nd)
```

Puede que ya veamos cuál es el problema que presenta este código, pero imaginemos que hemos escrito el resto del código del simulador (que podría ocupar miles de líneas de código) son hberlo detectado. Cuando ejecutamos el programa no falla, pero veremosque los coches virtuales se estrellan unos con otros.

Como ya hemos escrito el resto del programa, no tenemos ni idea dónde podría estar el error: Puede que esté en el código que simula los coches, en el código que simula a los conductores virtuales, etc. Podríamos tardar horas en rastrear la fuente del error está, de hecho, en la función `switchLights()`.

Si en vez de eso hubiésemos añadido en la función `switchLights()` una afirmación para comprobar que al menos una de las luces de un semáforo siempre esté en rojo, podríamos haber detectado rápidamente el problema. Añade lo siguiente al final de la función `switchLights()`:

```
assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
```

Con esta afirmación, nuestro programa terminaría lanzando un mensaje de error como el siguiente:

```
Traceback (most recent call last):
  File "C:/Users/Usuario/Dropbox/PYTHON/PROGRAMAS/traffic_lights_simulator.py",
line 14, in <module>
    switchLights(market_2nd)
  File "C:/Users/Usuario/Dropbox/PYTHON/PROGRAMAS/traffic_lights_simulator.py",
line 12, in switchLights
    assert 'red' in stoplight.values(), 'Neither light is red! ' +
str(stoplight)
AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}
```

La línea más importante de este error es la última: La afirmación ha detectado que ninguna de las direcciones del semáforo estaba en rojo, lo que significa que el tráfico en la intersección podría venor desde cualquier dirección, lo quenprovocaría una colisión.

DESACTIVAR LAS AFIRMACIONES.

Podemos desactivar las afirmaciones pasando la opción `-O` al ejecutar el programa desde la ventana de comandos. Esta posibilidad es útil cuando ya hemos terminado de escribir y de testear nuestro programa, y no queremos ralentizarlo realizando este tipo de comprobaciones. Las afirmaciones se usan en fase de dasarrollo, y no en el producto final. En el momento en el que ya hayamos entregado el programa a nuestro cliente, éste debería funcionar perfectamente y sin errores, y las afirmaciones ya no son necesarias.

11.5. REGISTROS (LOGGINGS).

PARTE II: APLICACIONES (1). PROGRAMACIÓN DE VIDEOJUEGOS.

12. EL JUEGO DE LA PIZZA.

13. INVASIÓN ALIEN (1).

14. INVASIÓN ALIEN (2).

15. INVASIÓN ALIEN (3).

16. ASTERIODES.

PARTE III: APLICACIONES (2). AUTOMATIZACIÓN DE TAREAS.

17. BÚSQUEDA DE PATRONES CON EXPRESIONES REGULARES.

Probablemente estemos familiarizados con la búsqueda de textos, presionando CTRL+F (ó CTRL+B) y escribiendo las palabras que estamos buscando. Las *expresiones regulares* van un paso más allá: Nos permiten especificar un *patrón* de texto a buscar. Por ejemplo, puede que no nos sepamos el número de teléfono exacto de un cliente, pero si vive en los Estados Unidos o en Canadá, sabemos que tendrá tres dígitos seguidos de un guión, y después otros cuatro dígitos más (y opcionalmente, tres dígitos al principio para el código del área). De esta forma sabemos que $415-555-1234$ es un número de teléfono, y que $4{,}155{,}551{,}234$ no lo es.

Las expresiones regulares son útiles, pero no muchos programadores las conocen, a pesar de que la mayoría de editores de texto modernos, como Microsoft Word y OpenOffice, incluyen herramientas para buscar y para buscar/reemplazar que están basadas en expresiones regulares. Las expresiones regulares permiten ahorrar un montón de tiempo, no solo a los usuarios de software, sino también a los programadores. De hecho, el escritor técnico Cory Doctorow afirma que antes de enseñarse programación, deberían enseñarse las expresiones regulares:

> "Conocer las expresiones regulares puede significar la diferencia entre resolver un problema en 3 pasos o en 3000. Los expertos suelen olvidar que los problemas que pueden resolverse con un par de pulsaciones de teclas suelen llevarles a otras personas días y días de tedioso trabajo".

En este capítulo empezaremos escribiendo un programa de búsqueda de patrones de texto *sin* usar expresiones regulares, y a continuación, veremos cómo usar expresiones regulares para hacer lo mismo con un programa mucho más corto y sencillo. Presentaremos los conceptos básicos de la correspondencia de patrones con expresiones regulares, y también estudiaremos algunas características más avanzadas, como la sustitución de cadenas y la creación de nuestras propias clases de caracteres. Al final del capítulo escribiremos un programa que extrae automáticamente los números de teléfono y las direcciones de correo electrónico de un documento de texto.

17.1. ENCONTRAR PATRONES DE TEXTO SIN EXPRESIONES REGULARES.

Imaginar que queremos hallar un número de teléfono en una cadena. Si el número es de los Estados Unidos, sabemos cuál es el patrón: tres números, un guión, tres números, un guión, y cuatro números. (Por ejemplo, $415-555-4242$).

Vamos a usar un función `isPhoneNumber()` para comprobar si una cadena se ajusta a este patrón, devolviendo a bien `True` o bien `False`. Abre una nueva ventana en el editor de archivos. escribe el siguiente código, y guárdalo como `isPhoneNumber.py`:

```
def isPhoneNumber(text):
❶   if len(text) != 12:
        return False  # not phone number-sized
    for i in range(0, 3):
❷       if not text[i].isdecimal():
            return False  # not an area code
❸   if text[3] != '-':
        return False  # does not have first hyphen
    for i in range(4, 7):
❹       if not text[i].isdecimal():
            return False  # does not have first 3 digits
```

```
    if text[7] != '-':
        return False  # does not have second hyphen
    for i in range(8, 12):
        if not text[i].isdecimal():
            return False  # does not have last 4 digits
    return True  # "text" is a phone number!

print('415-555-4242 is a phone number:')
print(isPhoneNumber('415-555-4242'))
print('Moshi moshi is a phone number:')
print(isPhoneNumber('Moshi moshi'))
```

Cuando este programa se ejecuta, la salida es:

```
415-555-4242 is a phone number:
True
Moshi moshi is a phone number:
False
```

La función `isPhoneNumber()` incluye un código que hace varias comprobaciones para ver si la cadena `text` contiene un número de teléfono válido. Si cualquiera de estas comprobaciones falla, la función devuelve `False`. (1) En primer lugar, el código comprueba si la cadena contiene exactamente 12 caracteres. (2) A continuación, comprueba que el código de área del número (esto es, los tres primeros caracteres en `text`) consiste únicamente en caracteres numéricos. El resto de la función comprueba que la cadena sigue el patrón de un número telefónico estadounidense: (3) El número debe tener el primer guión después del código de área, (4) tres dígitos numéricos, (5) otro guión, (6) y finalmente cuatro números más. Si la ejecución del programa consigue pasar todas las comprobaciones, la función devuelve `True` (7).

La llamada a `isPhoneNumber()` con el argumento `'415-555-4242'` devuelve `True`. La llamada a `isPhoneNumber()` con el argumento `'Moshi moshi'` devuelve `False`; la primera comprobación falla porque la cadena `'Moshi moshi'` no tiene 12 caracteres.

Tendríamos que añadir aún más código para encontrar este patrón de texto en una cadena más larga. Reemplaza las últimas cuatro llamadas a la función `print()` en `isPhoneNumber()` con el siguiente código:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
for i in range(len(message)):
    chunk = message[i:i+12]
    if isPhoneNumber(chunk):
        print('Phone number found: ' + chunk)
print('Done')
```

Cuando este programa se ejecute, la salida se parecerá a ésta:

```
Phone number found: 415-555-1011
Phone number found: 415-555-9999
Done
```

(1) Cada iteración del bucle `for` asigna un nuevo trozo (chunk) de 12 caracteres de la cadena `message` a la variable `chunk`. Por ejemplo, en la primera iteración `i` es `0`, y a `chunk` se le asigna `message[0:12]` (esto es, la cadena `'Call me at 4'`). En la siguiente iteración `i` es `1`, y a `chunk` se le asigna `message[1:13]` (la cadena `'all me at 41'`). De esta forma, vamos moviendo una "ventana" de tamaño 12 caracteres a lo largo de la cadena `message` en busca de un número telefónico.

(2) Además, en cada iteración le pasamos `chunk` a `isPhoneNumber()` para ver si ese trozo de 12 caracteres se ajusta al patrón buscado, y si es así, imprimimos `chunk`.

El bucle continúa moviéndose a lo largo de la cadena `message`, hasta que los 12 caracteres contenidos en `chunk` sean un número telefónico. El bucle recorre toda la cadena, comprobando cada trozo de 12 caracteres e imprimiendo cualquier trozo que satisfaga `isPhoneNumber()`. Una vez recorrida toda la cadena `message`, imprime `Done` (terminado).

Aunque en este ejemplo la cadena en `message` era corta, podría tener millones de caracteres, y este programa seguiría ejecutándose correctamente en menos de un segundo. Nuestro objetivo ahora es escribir un programa similar mediante expresiones regulares. Este programa también se ejecutará en menos de un segundo, pero las expresiones regulares nos permitirán que sea mucho más corto y sencillo.

17.2. ENCONTRAR PATRONES DE TEXTO CON EXPRESIONES REGULARES.

El programa previo para encontrar números de teléfono funciona, pero usa un montón de código para hacer una tarea muy limitada: La función `isPhoneNumber()` ocupa 17 líneas, pero solo es capaz de encontrar un patrón muy específico de números telefónicos. ¿Qué pasaría si el número telefónico tuveiese un formato como 415.555.4242 o (415) 555 – 4242? ¿Y si el número de teléfono tuviese una extensión, como 415 – 555 – 4242 x99? La función `isPhoneNumber()` fallaría a la hora de identificarlos. Podríamos añadir aún más código para detectar estos patrones adicionales, pero hay una forma más fácil de hacerlo.

Las **expresiones regulares** (denominadas **regex** en programación) son descripciones para un patrón de texto. Por ejemplo, un `\d` en una regex representa un carácter tipo dígito, esto es, un número de 0 a 9. La regex `\d\d\d-\d\d\d-\d\d\d\d` se usa en Python para identificar el mismo texto que identificaba la función `isPhoneNumber()`: Una cadena de tres números, un guión, tres números más, un guión, y otros cuatro números. Una cadena distinta nunca se ajustará a al regex `\d\d\d-\d\d\d-\d\d\d\d`.

Pero las expresiones regulares pueden ser mucho más sofisticadas. Por ejemplo, un 3 entre llaves (esto es, `{3}`) detrás de un patrón es como decir "Ajústate a este patrón tres veces". Por consiguiente, la regex `\d{3}-\d{3}-\d{4}` también se corresponde con el formato correcto de números telefónicos que buscaba la función `isPhoneNumber()`.

CREAR OBJETOS REGEX.

Todas las funciones regex disponibles en Python están en el módulo `re`. Escribe lo siguiente en el shell interactivo para importar este módulo:

```
>>> import re
```

> NOTA: La mayoría de los ejemplos de este capítulo necesitarán el módulo `re`, así que acuérdate de importarlo al principio de cualquier programa que escribas o siempre que reinicies el IDLE. En caso contrario, obtendrás un mensaje de error `NameError: name 're' is not defined`.

Si le pasamos un valor de tipo cadena que represente nuestra expresión regular a la función `re.compile()` obtenemos un objeto `Regex` que encarna el patrón de texto indicado en la expresión regular.

Para crear un objeto Regex que se ajuste al patrón de un número telefónico, escribe lo siguiente en el shell interactivo. (Recuerda que `\d` representa un carácter tipo dígito y que `\d\d\d-\d\d\d-\d\d\d\d` es la expresión regular para el patrón de un número telefónico correcto).

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

A partir de este momento, la variable `phoneRegex` contiene un objeto `Regex`.

Pasar cadenas en bruto a `re.compile()`.

Recordemos que los caracteres de escape en Python usan la barra inversa (`\`). El valor tipo cadena `'\n'` representa un carácter de cambio de línea, y no una barra seguida de una `n` minúscula. Para imprimir una barra inversa, hemos de usar el carácter de escape `\\`. Por consiguiente, `'\\n'` es la cadena que representa una barra inversa seguida de una `n` minúscula. Sin embargo, poniendo una `r` delante de la primera comilla de la cadena podemos indicar que la cadena es una *cadena en bruto*, la cual no interpretará los caracteres de escape.

Como las expresiones regulares suelen usar barras inversas, es conveniente pasar cadenas en bruto a la función `re.compile()`, en vez de pasarle cadenas como barras inversas dobles. Esto es muy útil porque es mucho más cómodo escribir `r'\d\d\d-\d\d\d-\d\d\d\d'` que `'\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d'`.

COINCIDENCIA CON OBJETOS REGEX.

El método `search()` de un objeto `Regex` busca en la cadena pasada como argumento cualquier coincidencia con la expresión regular que representa. El método `search()` devolverá `None` si el patrón regex no se encuantra en la cadena. Si ese patrón sí que está en la cadena, el método `search()` devolverá un objeto tipo `Match`. Los objetos `Match` tienen un método `group()` que devolverá el texto que coincide con el patrón buscado en la cadena original. (En breve explicaremos qué son los grupos). Por ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> import re
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

El nombre de variable `mo` es simplemente un nombre habitual para designar a los objetos `Match`. Al principio este ejemplo puede parecernos complicado, pero sin duda es mucho más corto el programa `isPhoneNumber.py`, y hace exactamente lo mismo.

Aquí, le pasamos el patrón deseado a `re.complite()` y almacenamos el objeto `Regex` resultante en la variable `phoneNumRegex`. A continuación llamamos al método `search()` sobre el objeto `phoneNumRegex` y le pasamos la cadena en la cual queremos buscar coincidencias con el patrón proporcionado. El resultado de la búsqueda se guarda en la variable `mo`. En este ejemplo, sabemos que el patrón se encuentra en la cadena, por lo que sabemos que `search()` devolverá un objeto `Match`. Sabiendo que `mo` contiene un objeto `Match` y no el valor `None`, podemos llamar al método `group()` sobre `mo` para devolver las coincidencias encontradas. Escribiendo `mo.group()` dentro de nuestra sentencia `print` imprimimos las coincidencias encontradas, en este caso $415 - 555 - 4242$.

REPASO DE LA BÚSQUEDA DE COINCIDENCIAS CON EXPRESIONES REGULARES.

Aunque hay varios pasos que dar a la hora de usar expresiones regulares, todos ellos son bastante sencillos:

1) Importamos el módulo regex con `import re`.
2) Creamos un objeto `Regex` con la función `re.compile()`. (Recordar que debemos usar una cadena en bruto).
3) Pasamos la cadena sobre la que queremos buscar al método `search()` del objeto `Regex`. Esto devolverá un objeto `Match` si el patrón buscado se encuentra en la cadena pasada.
4) Llamamos al método `group()` sobre el objeto `Match` para devolver una cadena con el texto que se ajusta al patrón buscado.

   NOTA: Aunque es recomendable escribir el código del ejemplo en el shell interactivo, también deberíamos hacer uso de los analizadores web de expresiones regulares, los cuales muestran la forma en la que se buscan coincidencias con una expresión regular en el texto que hayamos introducido. Un analizador regex recomendable es http://regexpal.com/.

17.3. MÁS SOBRE LA BÚSQUEDA DE PATRONES CON EXPRESIONES REGULARES.

Ahora que conocemos los pasos básicos para crear y encontrar objetos de expresiones regulares en Python, estamos listos para probar algunas funciones más avanzadas de búsqueda de patrones.

AGRUPACIÓN CON PARÉNTESIS.

Suponer que queremos separar el código de área del resto del número de teléfono. El uso de paréntesis nos permite **grupos** en la expresión regular: `(\d\d\d)-(\d\d\d-\d\d\d\d)`. Entonces, podemos usar el método `group()` del objeto `Match` generado para coger el texto coincidente de solo un grupo.

El primer conjunto de parántesis en la cadena regex será el grupo `1`. El segundo conjunto será el grupo `2`. Así, si le pasamos el entero `1` o `2` al método `group()` podemos obtener las diferentes partes del texto que coincide con el patrón regex proporcionado. Si le pasamos `0` o nada el método `group()` obtenemos todo el texto coincidente.

Escribe lo siguiente en el shell interactivo:

```
>>> import re
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

Si queremos recuperar todos los grupos de una sola tacada, usamos el método `groups()`. Notar que es la forma en plural del método `group()`.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

Como `mo.groups()` devuelve una tupla de múltiples valores, podemos usar el truco de las asignaciones múltiples para asignar cada valor a una variable independiente, como en la instrucción `areaCode, mainNumber = mo.groups()`.

Los paréntesis tienen un significado especial en las expresiones regulares, a saber, marcar los grupos. ¿Pero qué hacemos si necesitamos buscar paréntesis en nuestro texto? Por ejemplo, puede que los números de teléfono que estamos buscando tengan un código de área entre paréntesis. En este caso, debemos usar los caracteres de escape para `(` y `)` usando la barra inversa. Escribe lo siguiente en el shell interactivo:

```
>>> phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

Los caracteres de escape `\(` y `\)` en la cadena en bruto pasada a `re.complite()` buscarán los caracteres de paréntesis `(` y `)` en el texto a cotejar.

BÚSQUEDA DE MÚLTIPLES GRUPOS CON LA BARRA VERTICAL.

Al carácter `|` (accesible mediante ALT GR + tecla del 1) se le denomina *pipe* (tubo o tubería). Podemos usarlo allá donde necesitemos para buscar coincidencias con una o varias expresiones. Por ejemplo, la expresión regular `r'Batman|Joker'` hallará una coincidencia tanto con `'Batman'` como con `'Joker'`.

Cuando *tanto* Batman como Joker se encuentran en la cadena donde estamos buscando, se devuelve un objeto Match que alberga la primera coincidencia de la expresión regular en el texto. Escribe lo siguiente en el shell interactivo:

```
>>> import re
>>> heroRegex = re.compile(r'Batman|Joker')
>>> mo1 = heroRegex.search('Batman and Joker live in Gotham.')
>>> mo1.group()
'Batman'
>>> mo2 = heroRegex.search('Joker and Batman fight in Gotham.')
>>> mo2.group()
'Joker'
```

> NOTA: Podemos encontrar todas las coincidencias con la expresión regular mediante el método `findall()`, del cual hablaremos más adelante.

También podemos usar el carácter pipe para buscar coincidencias con uno de entre varios patrones en nuestra expresión regular. Por ejemplo, imaginar que queremos buscar coincidencias con una cualquiera de las cadenas `'Batman'`, `'Batmobile'`, `'Batcopter'`, y `'Batbat'`. Como todas estas cadenas comienzan con `Bat`, sería interesante si pudiésemos especificar este prefijo una sola vez. Esto puede hacerse con los paréntesis. Escribe lo siguiente en el shell interactivo:

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

La llamada al método `mo.group()` devuelve la coincidencia completa `'Batmobile'`, mientras que `mo.group(1)` solo devuelve la parte de la coincidencia dentro del primer grupo de paréntesis, a saber, `'mobile'`. Usando el carácter pipe y agrupando mediante paréntesis, podemos especificar varios patrones alternativos en nuestra expresión regular.

Si en alguna aplicación queremos buscar coincidencias que incluyan al propio carácter pipe, debemos usar el carácter de escape `\|`.

BÚSQUEDAS OPCIONALES CON EL SÍMBOLO DE INTERROGACIÓN.

En ocasiones tendremos un patrón que querremos buscar sólo de forma opcional. El carácter `?` marca el grupo que lo precede como una parte opcional del patrón. A modo de ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

La parte `(wo)?` De la expresión regular significa que el patrón `wo` es un grupo opcional. La expresión regular buscará una coincidencia que tenga cero o una ocurrencias de `wo` dentro de ella. Ésta es la razón por la que la expresión regular encuentra `'Batwoman'` y `'Batman'`.

Volviendo al ejemplo del número de teléfono, podemos hacer que la expresión regular busque números de teléfono que incluyan o no incluyan un código de área. Escribe lo siguiente en el shell interactivo:

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'
>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242'
```

De nuevo, si necesitamos buscar coincidencias que incluyan al propio carácter `?`, debemos usar el carácter de escape `\?`.

BUSCAR CERO O MÁS COINCIDENCIAS CON EL ASTERISCO.

El símbolo `*` (llamado asterisco) significa "busca cero o más coincidencias", esto es, el grupo que precede al asterisco puede aparecer un número de veces cualquiera en el texto donde buscamos. Puede estar totalmente ausente o repetirse una y otra vez. Volvamos al ejemplo de Batman:

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

Para `'Batman'`, la parte `(wo)*` de la expresión regular encuentra cero instancias de `wo` en la cadena; para `'Batwoman'`, la parte `(wo)*` encuentra una instacia de `wo`; y para `'Batwowowowoman'`, la parte `(wo)*` encuentra cuatro instancias de `wo`.

Si necesitamos buscar coincidencias que incluyan al asterisco, en la expresión regular debemos preceder el asterisco con la barra inversa para formar el carácter de escape `\*`.

BUSCAR UNA O MÁS COINCIDENCIAS CON EL SIGNO MÁS.

Mientras que `*` significa "busca cero o más coincidencias", el `+` (signo más) significa "busca una o más coincidencias". Al contrario que el asterisco, que no requiere que su grupo aparezca en la cadena coincidente, el grupo que precede a un signo más debe aparecer *al menos una vez*. Y esto no es opcional. Escribe lo siguiente en el shell interactivo, y compara con las expresiones regulares con asterisco del la sección previa:

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'
>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'
>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

La expresión regular `Bat(wo)+man` no encontrará ninguna coincidencia en la cadena `'The Adventures of Batman'` ya que el signo más obliga a encontrar al menos un `wo`.

Si necesitamos buscar en una cadena el carácter del signo más, en la expresión regular debemos preceder el signo más con la barra inversa para formar el carácter de escape `\+`.

BUSCAR REPETICIONES ESPECÍFICAS CON LAS LLAVES.

Si tenemos un grupo que queremos que se repita un cierto número de veces, en nuestra expresión regular debemos poner detrás de ese grupo un número entre llaves. Por ejemplo, la expresión regular `(Ha){3}` buscará la cadena `'HaHaHa'`, pero no la cadena `'HaHa'`, ya que esta última solo tiene dos repeticiones del grupo `(Ha)`.

En vez de poner un número, podemos especificar un rango escribiendo un mínimo, una coma, y un máximo entre llaves. Por ejemplo, la expresión regular `(Ha){3,5}` buscará `'HaHaHa'`, `'HaHaHaHa'`, y `'HaHaHaHaHa'`.

También podemos dejar el primer o el segundo número sin poner dentro de las llaves, para indicar que el mínimo o el máximo no está limitado. Por ejemplo, `(Ha){3,}` buscará tres o más instancias del grupo `(Ha)`, mientras que `(Ha){,5}` buscará de cero a cinco instancias. Las llaves nos pueden ayudar a hacer nuestras expresiones regulares más cortas. Estas dos expresiones regulares buscarán el mismo patrón:

```
(Ha){3}
(Ha)(Ha)(Ha)
```

Y estas dos expresiones regulares también buscarán exactamente el mismo patrón:

```
(Ha){3,5}
((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))
```

Escribe lo siguiente en el shell interactivo:

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'
>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

Aquí, `(Ha){3}` encuentra una coincidencia en `'HaHaHa'` pero no en `'Ha'`. Como no hay una coincidencia en `'Ha'`, `search()` devuelve `None`.

17.4. BÚSQUEDA VORAZ Y NO VORAZ.

Escribe lo siguiente en el shell interactivo:

```
>>> haRegex = re.compile(r'(Ha){3,5}')
>>> mo = haRegex.search('HaHaHaHaHa')
>>> mo.group()
'HaHaHaHaHa'
```

Como `(Ha){3,5}` puede encontrar tres, cuatro, o cinco instancias de `Ha` en la cadena `'HaHaHaHaHa'`, tal vez nos preguntemos por qué razón la llamada al método `group()` del objeto `Match` devuelve `'HaHaHaHaHa'` en vez de devolver otras posibilidades más cortas. Después de todo, `'HaHaHa'` y `'HaHaHaHa'` también son coincidencias básicas de la expresión regular `(Ha){3,5}`.

Las expresiones regulares en Python son **voraces** por definición, lo que significa que en situaciones ambiguas como la del ejemplo, siempre encontrarán la cadena más larga posible. La versión **no voraz** del uso de llaves para encontrar la cadena más cota posible utiliza la llave de cierre junto con un símbolo de interrogación.

Escribe lo siguiente en el shell interactivo, y observa la diferencia entre las formas voraz y no voraz del uso de las llaves a la hora de buscar en la misma cadena:

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'
>>>
>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Notar que en las expresiones regulares, el símbolo de interrogación puede tener dos significados bien distintos: declarar una búsqueda no voraz, o marcar un grupo opcional. Estos dos significados están totalmente desvinculados.

17.5. EL MÉTODO FINDALL().

Además del método `search()`, los objetos `Regex` también disponen del método `findall()`. Mientras que `search()` devuelve un objeto `Match` del *primer* objeto coincidente con el patrón de la expresión regular que encuentre en la cadena, el método `findall()` devolverá las cadenas de todas las coincidencias encontradas en la cadena donde está buscando. Para constatar que `search()` solo devuelve un objeto `Match` de la primera instancia del texto coincidente, escribe lo siguiente en el shell interactivo:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')
>>> mo.group()
'415-555-9999'
```

Por otro lado, `findall()` no devolverá un objeto `Match`, sino una lista de cadenas, siempre que no haya grupos en la expresión regular. Cada cadena de la lista es un trozo del texto buscado que implica una coincidencia con la expresión regular. Escribe lo siguiente en el shell interactivo:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

En el caso de que haya grupos en la expresión regular, `findall()` devolverá una lista de tuplas. Cada tupla representa una coincidencia encontrada, y sus objetos son las cadenas encontradas para cada grupo de la expresión regular. Para ver `findall()` en acción, escribe lo siguiente en el shell interactivo (observa que la expresión regular que se está compilando tiene ahora grupos en paréntesis):

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '9999'), ('212', '555', '0000')]
```

Para resumir lo que devuelve el método `findall()`, recordemos que:

1) Cuando lo llamamos sobre una expresión regular sin grupos, como por ejemplo `\d\d\d-\d\d\d-\d\d\d\d`, el método `findall()` devuelve una lista con todas las cadenas encontradas, como `['415-555-9999', '212-555-0000']`.
2) Cuando lo llamamos sobre una expresión regular que tiene grupos, como por ejemplo `(\d\d\d)-(\d\d\d)-(\d\d\d\d)`, el método `findall()` devuelve una lista de tuplas de cadenas () una cadena por cada grupo) como `[('415', '555', '1122'), ('212', '555', '0000')]`.

17.6. CLASES DE CARÁCTER.

En el primer ejemplo de expresión regular sobre números telefónicos aprendimos que `\d` podría representar un dígito numérico cualquiera. Es decir, `\d` es una abreviatura de la expresión regular `(0|1|2|3|4|5|6|7|8|9)`. Hay muchas más **clases de carácter** que podemos usar como abreviaturas, como las mostradas en la tabla:

Shorthand character class	Represents
\d	Any numeric digit from 0 to 9.
\D	Any character that is *not* a numeric digit from 0 to 9.
\w	Any letter, numeric digit, or the underscore character. (Think of this as matching "word" characters.)
\W	Any character that is *not* a letter, numeric digit, or the underscore character.
\s	Any space, tab, or newline character. (Think of this as matching "space" characters.)
\S	Any character that is *not* a space, tab, or newline.

Las clases de carácter son útiles para acortar las expresiones regulares. La clase de carácter `[0-5]` solo encontrará los números del 0 al 5; esto es mucho más corto que escribir `(0|1|2|3|4|5)`.

A modo de ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids,
7 swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans',
'6 geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

La expresión regular `\d+\s\w+` encontrará todo texto que tenga uno o más dígitos numéricos (`\d+`), seguidos de un carácter de espacio en blanco (`\s`), seguido de una o más caracteres de tipo letra, dígito, o guión bajo (`\w+`). El método `findall()` devuelve una lista con todas las cadenas que coincidan con el patrón de esta expresión regular.

17.7. CONSTRUIR NUESTRAS PROPIAS CLASES DE CARÁCTER.

Hay veces en las que queremos buscar coincidencias con un conjunto de caracteres, pero las abreviaturas basadas en clases de carácter (`\d`, `\w`, `\s`, etc.) son demasiado largas. En tales casos, podemos definir nuestras propias clases de carácter usando corchetes. Por ejemplo, la clase de carácter `[aeiouAEIOU]` buscará cualquier vocal, tanto minúscula como mayúscula.

Escribe lo siguiente en el shell interactivo:

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

También podemos incluir rangos de letras o números usando un guión. Por ejemplo, la clase de carácter `[a-zA-Z0-9]` buscará todas las letras minúsculas, todas las letras mayúsculas, y todos los números.

Notar que dentro de los corchetes los símbolos habituales de las expresiones regulares no se interpretan como tales. Esto significa que no tenemos que usar la barra inversa para producir los caracteres de escape de `.`, `*`, `?`, o `()`. Por ejemplo, la clase de carácter `[0-5.]` buscará dígitos del 0 al 5 y un punto. No hace falta que escribamos `[0-5\.]`.

Ubicando un acento circunflejo (^) justo después del corchete de apertura de una clase de carácter podemos construir una clase de carácter negativa. Una **clase de carácter negativa** buscará todos los caracteres que *no* estén en la clase de carácter. Por ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> consonantRegex = re.compile(r'[^aeiouAEIOU]')
>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')
['R', 'b', 'C', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.',
' ', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

Ahora, en lugar de buscar coincidencias con todas las vocales, estamos buscando cualquier carácter que no sea una vocal.

17.8. LOS CARACTERES ACENTO CIRCUNFLEJO Y DÓLAR.

También podemos usar el acento circunflejo (^) al principio de una expresión regular para indicar que la coincidencia debe ocurrir al comienzo del texto en el que estamos buscando. De la misma forma, podemos poner un dólar (`$`) al final de una expresión regular para indicar que la cadena debe terminar con el patrón indicado en la expresión regular. Y podemos usar `^` y `$` conjuntamente para indicar que la cadena completa debe coincidir con la expresión regular (es decir, para indicar que no basta con que la coincidencia se limite a un subconjunto de la cadena).

Por ejemplo, la expresión regular `r'^Hello'` busca coincidencias con cadenas que empiecen con `'Hello'`. Escribe lo siguiente en el shell interactivo:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<re.Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

La expresión regular `r'\d$'` busca cadenas que terminen con un carácter numérico del `0` al `9`. Escribe lo siguiente en el shell interactivo:

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<re.Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

La expresión regular `r'^\d+$'` busca cadenas que empiecen y terminen con uno o más caracteres numéricos. Escribe lo siguiente en el shell interactivo:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<re.Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12 34567890') == None
True
```

Las dos últimas llamadas a `search()` en el último ejemplo muestran que si usamos `^` y `$`, la cadena completa debe verificar la expresión regular.

17.9. EL CARÁCTER COMODÍN.

El carácter . (o punto) en una expresión regular se denomina comodín, y buscará coincidencias con cualquier carácter excepto el de nueva línea. A modo de ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Debemos recordar que el punto sólo buscará un carácter, y ésta es la razón por la que el texto flat en el ejemplo previo sólo se encontró como lat.

Como siempre, para buscar coincidencias con un auténtico punto, debemos usar el carácter de escape `\.`.

BUSCAR LO QUE SEA CON EL PUNTO - ASTERISCO.

En ocasiones querremos buscar coincidencias con cualquier cosa. Vamos a explicarnos: imaginar que queremos buscar la cadena `'First Name:'`, seguida de lo que sea, seguida de `'Last Name:'`, seguida de lo que sea. En esos casos, podemos usar el punto – asterisco (`.*`) para indicar ese "lo que sea". Recordemos que el punto significa "cualquier carácter excepto una nueva línea", y el asterisco significa "cero o más ocurrencias del carácter precedente".

Escribe lo siguiente en el shell interactivo:

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
>>> mo = nameRegex.search('First Name: Alejandro Last Name: Martínez')
>>> mo.group(1)
'Alejandro'
>>> mo.group(2)
'Martínez'
```

El punto – asterisco usa modo voraz: Siempre intentará encontrar tantas coincidencias como sea posible. Para operar de forma no voraz, debemos usar el punto, el asterisco, y el símbolo de interrogación (`.*?`). Como con las llaves, el símbolo de interrogación le dice a Python que busque de forma no voraz. Escribe lo siguiente en el shell interactivo para ver la diferencia entre las versiones voraz y no voraz:

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'
```

```
>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

Ambas expresiones regulares vienen a decir "busca un símbolo menor que (<), seguido de lo que sea, seguido de un símbolo mayor que (>)". Pero la cadena `'<To serve man> for dinner.>'` tiene dos posibles coincidencias para el símbolo mayor que. En la versión no voraz de la expresión regular, Python solo encuentra la cadena más corta posible, `'<To serve man>'`. En la versión voraz, Python encuentra la cadena más larga posible, `'<To serve man> for dinner.>'`.

BUSCAR SALTOS DE LÍNEA CON EL PUNTO.

El punto – asterisco buscará cualquier cosa excepto los saltos de línea. Pero pasando `re.DOTALL` como segundo argumento a `re.compile()` podemos hacer que el punto sea capaz de buscar coincidencias con todos los caracteres, incluyendo los de salto de línea. Escribe lo siguiente en el shell interactivo:

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'
>>>
>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.\n
Uphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

La expresión regular `noNewlineRegex`, que no incluye el argumento `re.DOTALL` en la llamada a `re.compile()` que la creó, buscará todas las coincidencias hasta llegar al primer carácter de sato de línea. Por su parte, `newlineRegex` sí que incluye el argumento `re.DOTALL`, busca todo. Ésta es la razón por la que la llamada `newlineRegex.search()` encuentra toda la cadena, incluyendo los caracteres de salto de línea.

17.10. REPASO DE LOS SÍMBOLOS REGEX.

En este capítulo hemos presentado un montón de notación, así que vamos a hacer un rápido repaso de todo lo que hemos visto:

- `?` busca cero o una coincidencias con el grupo al que precede.
- `*` busca cero o más coincidencias con el grupo al que precede.
- `+` busca una o una coincidencias con el grupo al que precede.
- `{n}` busca exactamente n coincidencias con el grupo al que precede.
- `{n,}` busca n o más coincidencias con el grupo al que precede.
- `{,m}` busca 0 ó m coincidencias con el grupo al que precede.
- `{n,m}` busca al menos n y como mucho m coincidencias con el grupo al que precede.
- `{n,m}?` ó `*?` ó `+?` Efectúa una búsqueda no voraz del grupo al que precede.
- `^spam` significa que la cadena debe comenzar con *spam*.
- `spam$` significa que la cadena debe terminar con *spam*.
- `.` busca cualquier carácter, excepto los caracteres de salto de línea.
- `\d`, `\w`, y `\s` buscan un carácter de tipo dígito, letra, y espacio en blanco, respectivamente.
- `\D`, `\W`, y `\S` buscan cualquier cosa excepto un carácter de tipo dígito, letra, y espacio en blanco, respectivamente.
- `[abc]` busca cualquier carácter que esté incluido entre los corchetes (*a*, *b*, o *c*, en este caso).
- `[^abc]` busca cualquier carácter que no esté incluido entre los corchetes.

17.11. BÚSQUEDA SIN DISTINCIÓN ENTRE MAYÚSCULAS Y MINÚSCULAS.

Normalmente, las expresiones regulares buscan coincidencias distinguiendo entre mayúsculas y minúsculas, según la expresión regular especificada. A modo de ejemplo, las siguientes expresiones regulares buacarán cadenas completamente distintas:

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

Pero en ocasiones deberemos buscar coincidencias sin prestar atención a si las letras están en mayúsculas o minúsculas. Para hacer que la expresión regular busque sin tener en consideración este hecho, a la función `re.compile()` debemos pasarle como segundo argumento `re.IGNORECASE` o `re.I`. Escribe lo siguiente en el shell interactivo:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'
>>>
>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'
>>>
>>> robocop.search('Alejandro, why does your programming book talk about
robocop so much?').group()
'robocop'
```

17.12. SUSTITUIR CADENAS CON EL MÉTODO SUB().

Las expresiones regulares no solo pueden encontrar patrones de texto, sino también sustituir texto nuevo en el lugar de los patrones encontrados. Para ello disponemos del método `sub()` de los objetos `Regex`, el cual recibe dos argumentos. El primer argumento es una cadena para reemplazar las coincidencias encontradas. El segundo es la cadena de la expresión regular. El método `sub()` devuelve una cadena con las sustituciones aplicadas.

A modo de ejemplo, escribe lo siguiente en el shell interactivo:

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents
to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'
```

En ocasiones podremos necesitar usar el texto encontrado como parte de la sustitución. En el primer argumento de `sub()`, podemos escribir `\1`, `\2`, `\3`, etc., para decir "Escribe el texto de los grupos 1, 2, 3, etc. en la sustitución".

Por ejemplo, imaginar que queremos censurar los nombres de los agentes secretos mostrando sólo las primeras letras de sus nombres. Para hacerlo, podríamos usar la expresión regular `Agent (\w)\w*` y pasar `r'\1****'` como primer argumento a `sub()`. El `\1` en la cadena será reemplazado por cualquier texto que coincida con el grupo 1, esto es, el grupo `(\w)` en la expresión regular.

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that
Agent Eve knew Agent Bob was a double agent.')
'A**** told C**** that E**** knew B**** was a double agent.'
```

17.13. MANEJO DE EXPRESIONES REGULARES COMPLEJAS.

Las expresiones regulares son relativamente sencillas cuando el patrón de texto a buscar es simple. Pero buscar patrones de texto complejos puede suponer expresiones regulares largas y enrevesadas. Podemos mitigar este problema diciéndole a la función `re.compile()` que ignore los espacios en blanco y comentarios dentro de la cadena de la expresión regular. Para activar este "modo verboso" debemos pasar la variable `re.VERBOSE` como segundo argumento a `re.compile()`.

De esta forma, el lugar de tener una expresión regular tan compleja como ésta:

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}
(\s*(ext|x|ext.)\s*\d{2,5})?)')
```

, podemos extenderla a lo largo de múltiples líneas con comentarios como sigue:

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?            # area code
    (\s|-|\.)?                    # separator
    \d{3}                         # first 3 digits
    (\s|-|\.)                     # separator
    \d{4}                         # last 4 digits
    (\s*(ext|x|ext.)\s*\d{2,5})?  # extension
    )''', re.VERBOSE)
```

Observa que el ejemplo usa sintaxis con comilla triples (`'''`) para crear una cadena multilínea, de forma que podamos extender la definición de la expresión regular a lo largo de varias líneas, haciéndola mucho más legible.

Las reglas que gobiernan los comentarios dentro de la cadena de la expresión regular son las mismas que aplican en el código Python habitual: El símbolo `#` y todo lo que le sigue hasta el final de la línea se ignora. Además, los espacios extra dentro de la cadena multilínea de la expresión regular no se consideran parte del patrón de texto a buscar. Esto nos permite organizar la expresión regular de forma que sea más fácil de leer.

17.14. COMBINAR re.IGNORECASE, re.DOTALL, Y re.VERBOSE.

¿Y si queremos usar `re.VERBOSE` para escribir comentarios en nuestras expresiones regulares, pero también queremos usar `re.IGNORECASE` para no distinguir entre mayúsculas y minúsculas? Desafortunadamente, la función `re.compile()` solo puede recibir un valor como segundo argumento. Pero podemos soslayar esta limitación combinando las variables `re.IGNORECASE`, `re.DOTALL`, y `re.VERBOSE` usando el carácter pipe (`|`), que en este contexto es conocido como el **operador bitwise or**.

Así pues, si queremos una expresión regular que no distinga entre mayúsculas y minúsculas, y que incluya búsquedas de saltos de línea con el punto, deberíamos formar nuestra llamada a `re.compile()` como:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

Al pasar las tres opciones como segundo argumento, la llamada a `re.compile()` queda como:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

Esta sintaxis está un poco anticuada y proviene de las primeras versiones de Python. Los detalles del uso de los operadores bitwise están más allá de nuestros objetivos en este texto. También podemos pasar otras opciones como segundo argumento a `re.compile()`, pero son poco comunes.[11]

17.15. PROYECTO: EXTRACTOR DE NÚMEROS TELEFÓNICOS Y DIRECCIONES DE CORREO.

17.16. EJERCICIOS DEL CAPÍTULO 17.

Ejercicio 17.1. Guarda el programa como `Ejer17.1.py`.

[11] Puedes echar un vistazo a los recursos en la web http://nostarch.com/automatestuff/ para más información sobre los operadores bitwise y las diferentes opciones para el segundo argumento de `re.compile()`.

18. ORGANIZAR ARCHIVOS.

19. OBTENER INFORMACIÓN DE LA WEB (WEB SCRAPING).

20. TRABAJAR CON HOJAS DE CÁLCULO EXCEL.

21. TRABAJAR CON DOCUMENTOS WORD Y PDF.

22. TRABAJAR CON ARCHIVOS CSV Y DATOS JSON.

23. CONTROL DEL TIEMPO, PLANIFICACIÓN DE TAREAS, Y LANZAMIENTO DE PROGRAMAS.

24. ENVÍO DE CORREOS ELECTRÓNICOS Y DE MENSAJES DE TEXTO.

25. MANIPULACIÓN DE IMÁGENES.

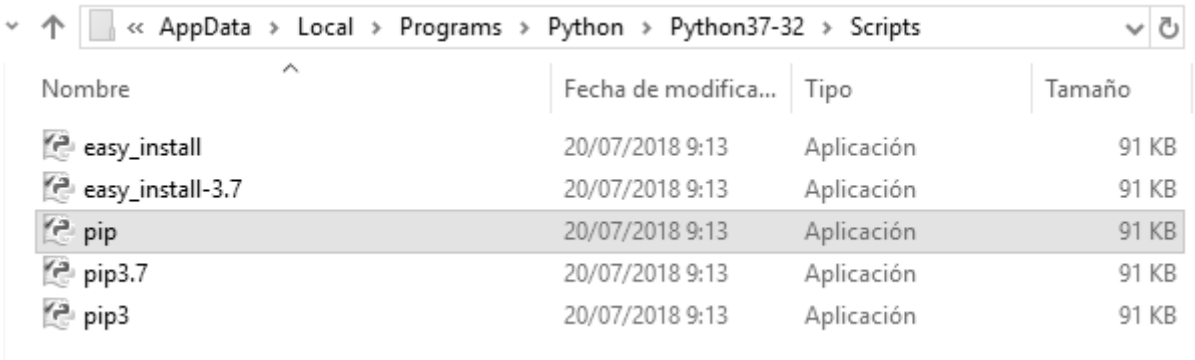
26. CONTROL DEL TECLADO Y DEL RATÓN CON AUTOMATIZACIÓN GUI.

ANEXO A: INSTALACIÓN DE MÓDULOS DE TERCEROS.

Más allá de los módulos estándar de Python, los desarrolladores han programado sus propios módulos para ampliar las capacidades de Python. La forma más habitual de instalar estos **módulos de terceros** es usar la herramienta **pip** de Python. Esta herramienta permite descargar e instalar de forma segura módulos de Python en nuestro ordenador, desde el sitio web de la **Python Software Foundation** (https://pypi.python.org/). El PyPI (esto es, el Python Package Index) es una especie de tienda gratuita de apps para módulos Python.

A.1. LA HERRAMIENTA PIP.

El archivo ejecutable para la herramienta pip se llama *pip* en Windows y *pip3* en OS X y Linux. En Windows, podremos encontrar el pip en *C*:\Users\Usuario\AppData\Local\Programs\Python\Python37-32\Scripts\pip.exe.[12]



A.2. INSTALACIÓN DE MÓDULOS DE TERCEROS.

La herramienta pip está diseñada para usarse desde la consola de comandos (símbolo del sistema). Para ello, y desde la ventana de comandos, acudimos a la carpeta donde está el programa pip.exe (usando los comandos `cd nombreCarpeta`, `cd..`, y `dir` para navegar por las carpetas), y lo ejecutamos pasándole el comando `install` seguido del nombre del módulo que queremos instalar. Por ejemplo, en Windows deberíamos escribir `pip install ModuleName`, donde `ModuleName` es el nombre del módulo (ver figura más abajo).

Si ya tenemos el módulo instalado en nuestro ordenador pero queremos actualizarlo a la última versión disponible en PyPI, debemos ejecutar `pip install -U ModuleName`.

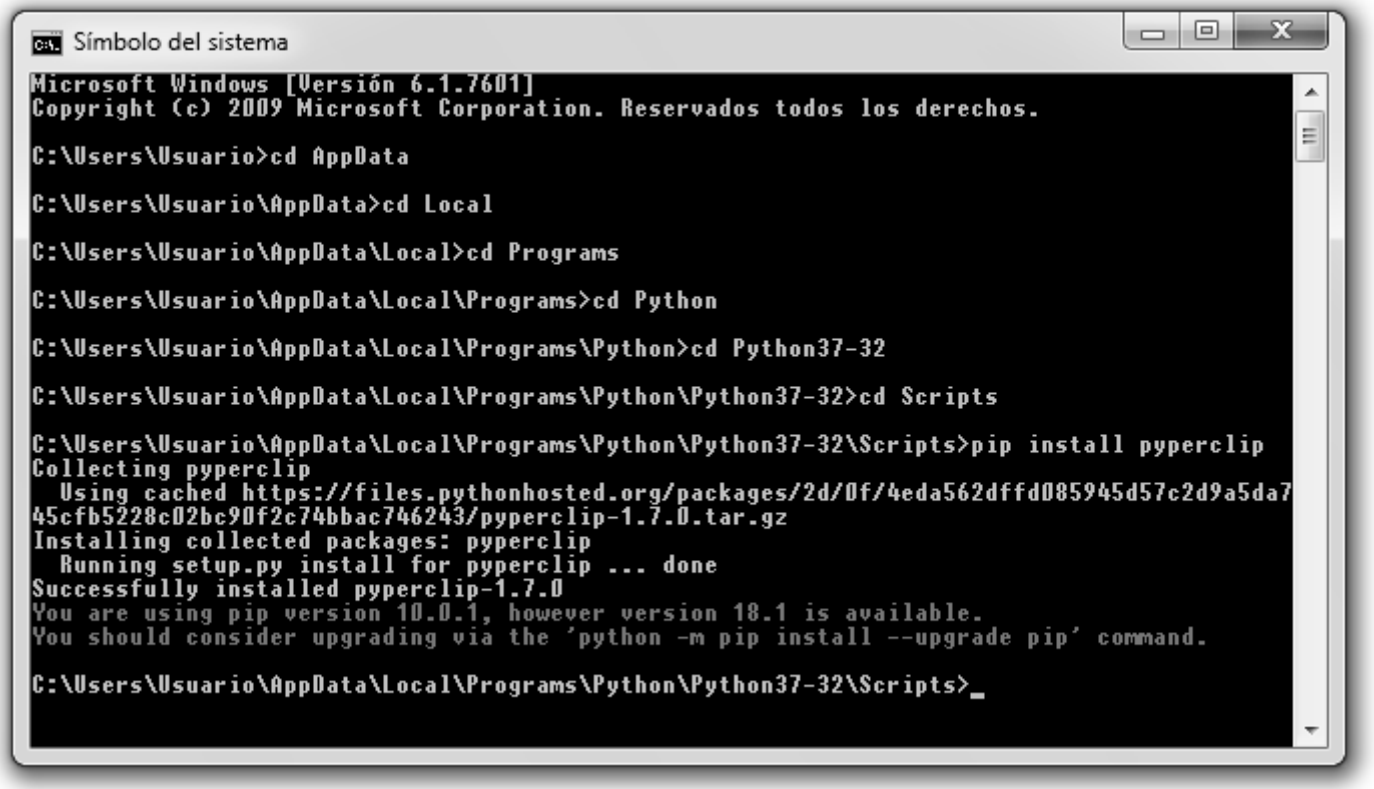
Después de instalar el módulo, podemos comprobar que está exitosamente instalado ejecutando `import ModuleName` en el shell interactivo. Si no se muestran mensajes de error, podemos asumir que el módulo se instaló correctamente.

Podemos instalar todos los módulos cubiertos en este libro ejecutando los comandos listados a continuación:

- `pip install pyperclip`.
- `pip install send2trash`.

[12] La ubicación del archivo pip.exe depende de cómo y dónde se instalase el Python en tu ordenador. Utiliza la herramienta de búsqueda para encontrar la carpeta de instalación de Python en tu equipo. (Asegúrate de visualizar las carpetas y archivos ocultos).

- pip install request.
- pip install beautifulsoup4.
- pip install selenium.
- pip install openpyxl.
- pip install PyPDF2.
- pip install pyhon-docx.
- pip install imapclient.
- pip install pyxmail.
- pip install twilio.
- pip install pillow.
- pip install pyautogui.



ANEXO B: EJECUCIÓN DE PROGRAMAS.

Si tenemos un programa en el editor de archivos del IDLE, para ejecutarlo basta con presionar la tecla F5 o con seleccionar el menú Run → Run Module. Esta es una forma sencilla de ejecutar programas mientras los estamos escribiendo, pero tener que abrir el IDLE para ejecutar programas que ya están terminados es un incordio. Afortunadamente, hay otras formas más convenientes de ejecutar programas Python.

B.1. LÍNEA SHEBANG.

La primera línea de todos nuestros programas Python debería ser una **línea shebang**, que sirve para decirle a nuestro ordenador que queremos que Python ejecute ese programa. La línea shebang siempre comienza con `#!`, pero el resto de la línea depende del sistema operativo instalado en nuestro equipo:

- En Windows, la línea shebang es `#! python3`.
- En OS X, la línea shebang es `#! usr/bin/env python3`.
- En Linux, la línea shebang es `#! usr/bin/ python3`.

Podemos ejecutar programas de Python desde el IDLE sin la línea shebang, pero esta línea es necesaria para poder ejecutarlos desde la consola de comandos (el símbolo de sistema en Windows).

B.2. EJECUTAR PROGRAMAS PYTHON EN WINDOWS.

En Windows, el intérprete de Python se localiza en C:\Users\Usuario\AppData\Local\Programs\Python\Python37-32\python.exe.[13] Alternativamente, el programa *py.exe* leerá la línea shebang al principio del código fuente del archivo .py, y ejecutará la versión apropiada de Python para ese código. El programa *py.exe* se asegurará de ejecutar el programa Python con la versión correcta de Python, en el caso de que haya múltiples versiones instaladas en nuestro ordenador.

Para facilitar la ejecución de programas Python, debemos crear un **archivo por lotes** .bat (batch file) para ejecutar programas Python con py.exe. Para construir un archivo por lotes, crea un nuevo archivo de texto que únicamente contenga la siguiente línea:

```
@py.exe C:\ruta\a\tu\programaPython.py %*
```

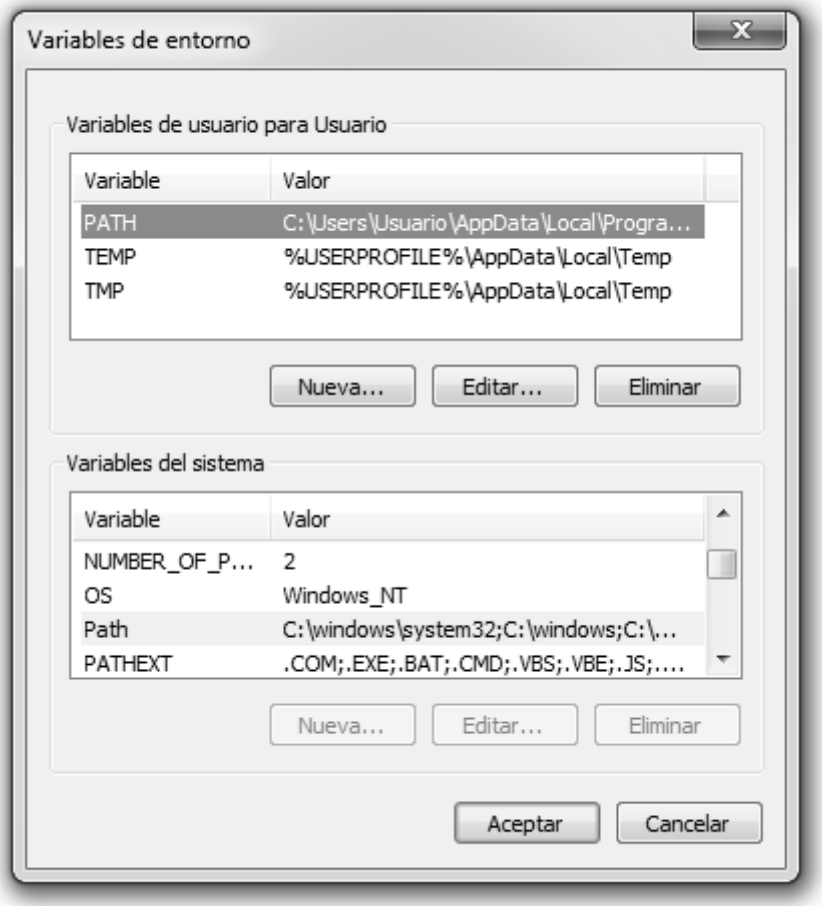
Reemplaza esta ruta por la ruta absoluta a tu programa, y guarda este archivo de texto con una extensión .bat (por ejemplo, *pythonScript.bat*). Este archivo por lotes evitará que tengamos que teclear toda la ruta del programa de Python cada vez que queramos ejecutarlo. Es recomendable guardar el archivo por lotes .bat y el programa de Python .py en la misma carpeta, como por ejemplo, C:\MisProgramasPython o C:\Users\YourName\MisProgramasPython.

Ahora, debemos añadir la ruta a esta carpeta al PATH de Windows, de forma que podamos ejecutar los archivos .bat desde la ventana de diálogo "ejecutar" de Windows. Para ello, hemos de modificar la **variable de entorno** PATH: Pinchamos en el menú de inicio y tecleamos "Editar las variables de entorno de esta cuenta". Esta opción debería autocompletarse después de que hayamos empezado a escribirla. La ventana para ajustar las variables de entorno debería parecerse a la de la figura.

En las variables de usuario, seleccionamos la variable PATH (o la creamos nueva, si no existe). En el campo de valor, añadimos un punto y coma (;) detrás de la última ruta que ya esté añadida, escribimos la ruta

[13] De nuevo, la ubicación del archivo python.exe depende de cómo y dónde esté instalado Python en tu ordenador. Utiliza la herramienta de búsqueda para encontrar la carpeta de instalación de Python en tu equipo.

absoluta a la carpeta donde estén nuestros archivos .py y .bat, y clicamos OK. A partir de ahora podremos ejecutar cualquier programa de Python que esté en esa carpeta, simplemente presionando WIN+R y escribiendo el nombre del programa .py.



Por ejemplo, imagina que nuestro programa se llama `pythonScript.py`, que su archivo batch asociado es `pythonScript.bat`, y que ambos archivos están alojados en la carpeta C:\Users\YourName\MisProgramasPython. Una vez añadida esta ruta a la variable PATH, para ejecutar este programa desde la ventana de "ejecutar" (WIN + R) solo necesitamos escribir `pythonScript.py`. Esto hará que se ejecute el archivo `pythonScript.bat`, gracias al cual nos ahorraremos tener que escribir el comando `py.exe C:\Users\YourName\MisProgramasPython\pythonScript.py` en la ventana "ejecutar".

REFERENCIAS.

R.1. LIBROS.

1) Automate The Boring Stuff with Python, de Al Sweigart. (Ed. No Starch Press).
2) Python Programming for the Absolute Beginner, de Michael Dawson (Ed. Cengage Learning).
3) Python Crash Course, de Eric Matthes. (Ed. No Starch Press).
4) Invent your Own Comouter Games with Python, de Al Sweigart. (Ed. No Starch Press).
5) A Primer on Scientific Programming with Python, de Hans Petter Langtangen. (Ed. Springer).

R.2. WEBS.

1) https://erlerobotics.gitbooks.io/erle-robotics-learning-python-gitbook-free/
2) https://learning-python.com/class/Workbook/
3) https://www.programiz.com/python-programming/first-program
4) http://interactivepython.org/runestone/static/thinkcspy/toc.html
5) http://buildingskills.itmaybeahack.com/book/python-2.6/html/index.html
6) http://openbookproject.net/thinkcs/python/english3e/index.html
7) https://www.w3resource.com/python/python-tutorial.php
8) https://www.w3resource.com/python-exercises/
9) http://usingpython.com/menu/
10) https://www.practicepython.org/
11) http://introtopython.org/
12) https://repl.it/@Python_Cobra/

R.3. OTROS.

1) http://hplgit.github.io/primer.html/doc/pub/class/class-readable.html
2) https://www.python-course.eu/python_tkinter.php
3) https://programacionpython80889555.wordpress.com/2018/06/03/creando-una-calculadora-con-interfaz-grafica-con-python-y-tkinter-1a-parte/
4) http://usingpython.com/simple-gui-programming/
5) https://tkdocs.com/tutorial/index.html

Copiar figuras al 115%. (Algunas figuras (dibujos) a tamaño 70%. Diagramas de flujo al 105%)
Fuente IDLE: `Courier New(11)`
14 espacios desde principio de hoja hasta título de la parte.