# Inventa tus propios juegos de computadora con Python
# 3ª edición

**Traducido por Alfredo Carella**

**Alejandro Pernin**

**Francisco Palm**

**Escrito por Al Sweigart**

Libro Version ES.0.1, ISBN XXX-XXXXXXTODO

Si ha descargado este libro de un torrente, probablemente fuera de fecha. Ir a http://inventwithpython.com/es para descargar la última versión..

*Por Caro, con más amor de lo que nunca supo que tenía.*

# Nota para los padres y los compañeros programadores

Gracias por leer este libro usted. Mi motivación para escribir que venía de una brecha que vi en la literatura de hoy para los niños interesados en aprender a programar. Empecé a programar en el lenguaje de programación BASIC con un libro similar es esta.

Durante el curso de escribir esto, me he dado cuenta de cómo una lengua moderna como la programación Python ha hecho mucho más fácil y versátil para una nueva generación de programadores. Python tiene una curva de aprendizaje suave sin dejar de ser una lengua seria utilizado por programadores profesionales.

La generación actual de libros de programación se dividen en dos categorías. En primer lugar, los libros que no enseñan programación tanto como "software de creación de juego" o un idiotizada lenguajes de programación para hacer "fácil", al punto que es que ya no la programación. O segundo, enseñaron la programación como un libro de texto de matemáticas: Principios y conceptos todos ellos con poca aplicación dan al lector. Este libro tiene un enfoque diferente: mostrar el código fuente de los juegos de la derecha en la delantera y explicar la programación de Principios de los ejemplos.

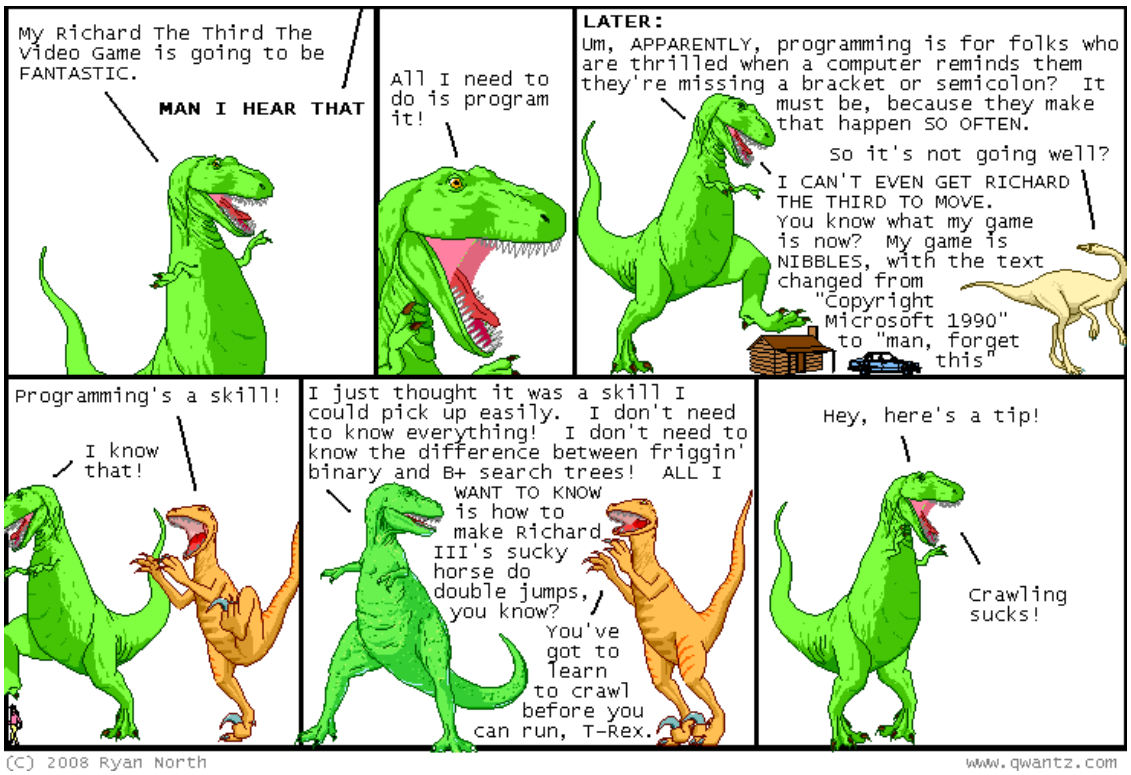
También he hecho este libro disponible bajo la licencia Creative Commons, que permite realizar copias y distribuir este libro (o fragmentos) con mi permiso completo, siempre y cuando la atribución que me queda intacto y se utiliza para fines no comerciales. (Vea la página de derechos de autor.) Quiero que este libro sea un regalo para un mundo que me ha dado tanto.

## ¿Qué hay de nuevo en la tercera edición?

La tercera edición presenta ningún nuevo contenido desde la segunda edición. Sin embargo, la tercera edición se ha simplificado que cubre el mismo contenido con un 20% menos páginas. Las explicaciones se han ampliado el cuando sea necesario y ambigüedades aclaró.

Capítulo 9 se dividió en capítulos 9 y 9 ½ para mantener la numeración de capítulos de la misma.

El código fuente ha sido intencionalmente mantenido la misma que la segunda edición es evitar la confusión. Si ya has leído la segunda edición, no hay razón para leer este libro. Sin embargo, si usted es nuevo en la programación, o la introducción de un amigo esta programación, esta tercera edición hará que el proceso sea más fácil, más suave y más divertido.




# ¿Quién es este libro?

La programación no es difícil. Pero es difícil encontrar materiales de aprendizaje que es para enseñarte cosas interesantes con la programación. Otros libros de ordenadores durante muchos temas programadores novatos puente no necesitan. Este libro le enseñará a programar sus propios juegos de ordenador. Vas a aprender un oficio útil y tener juegos divertidos para demostrarlo! Este libro es para:

- Los principiantes que quieren enseñar ordenador programación sí mismos, incluso si no tienen experiencia previa en programación.
- Los niños y adolescentes que quieren aprender a programar mediante la creación de juegos.
- Los adultos y los profesores que enseñan otros desean esta programación.
- Cualquier persona, joven o viejo, que quiere aprender a programar por el aprendizaje de un lenguaje de programación profesional.

# TABLE DE CONTENIDO

# Capítulo 1

# INSTALANDO PYTHON

Temas Tratados En Este Capítulo:
- Descargar e instalar el intérprete de Python
- Cómo usar este libro
- La página web de este libro en http://inventwithpython.com/es

¡Hola! Este libro te enseñará a programar creando videojuegos. Una vez que aprendas cómo funcionan los juegos en este libro, serás capaz de crear tus propios juegos. Todo lo que necesitas es una computadora, un software llamado el intérprete de Python, y este libro. El intérprete de Python es libre para descargar de Internet.

Cuando era niño, un libro como este me enseñó cómo escribir mis primeros programas y juegos. Era divertido y fácil. Ahora, siendo un adulto, sigo divirtiéndome programando y me pagan por hacerlo. Pero incluso si no te conviertes en un programador cuando crezcas, programar es una habilidad divertida y útil para tener.

Las computadoras son máquinas increíbles, y aprender a programarlas no es tan difícil como la gente cree. Si puedes leer este libro, puedes programar una computadora. Un **programa de computadora** es un conjunto de instrucciones que la computadora puede entender, igual que un libro de cuentos es un conjunto de oraciones que el lector entiende. Ya que los videojuegos no son más que programas de computadora, también están compuestos por instrucciones.

Para dar instrucciones a una computadora, escribes un programa en un lenguaje que la computadora comprende. Este libro enseña un lenguaje de programación llamado Python. Hay muchos otros lenguajes de programación, incluyendo BASIC, Java, JavaScript, PHP y C++.

Cuando era niño, era común aprender BASIC como un primer lenguaje. Sin embargo, nuevos lenguajes de programación tales como Python han sido inventados desde entonces. ¡Python es aún más fácil de aprender que BASIC! Pero sigue siendo un lenguaje de programación muy útil utilizado por programadores profesionales. Muchos adultos usan Python en su trabajo y cuando programan por diversión.

Los juegos que crearás a partir de este libro parecen simples comparados con los juegos para Xbox, PlayStation, o Nintendo. Estos juegos no tienen gráficos sofisticados porque están pensados para enseñar conceptos básicos de programación. Son deliberadamente sencillos de

modo que puedas enfocarte en aprender a programar. Los juegos no precisan ser complicados para ser divertidos.

## Descargar e Instalar Python

Necesitarás instalar un software llamado el intérprete de Python. **El programa intérprete** entiende las instrucciones que escribirás en lenguaje Python. De ahora en adelante me referiré al "software intérprete de Python" simplemente como "Python".

> **¡Nota importante!** Asegúrate de instalar Python 3, y no Python 2. Los programas en este libro usan Python 3, y obtendrás errores si intentas ejecutarlos con Python 2. Esto es tan importante que he agregado la caricatura de un pingüino en la Figura 1-1 para decirte que instales Python 3 así no te pierdes este mensaje.


Figura 1-1: Un pingüino extravagante te dice que instales Python 3.

Si usas Windows, descarga el instalador de Python (el archivo tendrá la extensión .msi) y haz doble clic sobre él. Sigue las instrucciones que el instalador muestra en pantalla:

1. Selecciona Instalar para Todos los Usuarios y haz clic en **Next** (Siguiente).
2. Elige *C:\Python34* como carpeta de instalación haciendo clic en **Next** (Siguiente).
3. Haz clic en **Next** (Siguiente) para omitir la sección de configuración de Python.

Si usas Mac OS X, descarga el archivo *.dmg* indicado para tu versión de OS X del sitio web y haz doble clic sobre él. Sigue las instrucciones que el instalador muestra en pantalla:

1. Cuando el paquete DMG se abra en una nueva ventana, haz doble clic sobre el archivo *Python.mpkg*. Es posible que necesites ingresar la clave de administrador.
2. Haz clic en **Continue** (Continuar) para pasar la sección Bienvenido y en **Agree** (Aceptar) para aceptar la licencia.
3. Selecciona HD Macintosh (o como sea que se llame tu disco rígido) y haz clic en **Install** (Instalar).

Si usas Ubuntu, puedes instalar Python del Centro de Software de Ubuntu siguiendo estos pasos:

1. Abre el Centro de Software de Ubuntu.
2. Escribe *Python* en el cuadro de búsqueda en la esquina superior derecha de la ventana.
3. Elige **IDLE (using Python 3.4)**, o la que sea la última versión en este momento.
4. Haz clic en **Install** (Instalar). Tal vez necesites la clave de administrador para completar la instalación.

## Iniciando IDLE

La sigla IDLE (**I**nteractive **D**eve**L**opment **E**nvironment en inglés) significa Entorno Interactivo de Desarrollo. El entorno de desarrollo es como un software procesador de palabras para escribir programas de Python. Iniciar IDLE es diferente para cada sistema operativo.

Sobre Windows, haz clic en el botón Inicio en la esquina inferior izquierda, teclea "IDLE" y selecciona **IDLE (Python GUI)**.

Sobre Mac OS X, abre la ventana de Finder y haz clic en **Applications**. Luego haz clic en **Python 3.4**. Luego clic sobre el ícono de IDLE.

Sobre Ubuntu o Linux, abre una terminal y teclea "idle3". También puede ser posible hacer clic en **Applications** en el borde superior de la pantalla. Luego haz clic sobre **Programming** y después **IDLE 3**.

La ventana que aparece la primera vez que ejecutas IDLE es **la consola interactiva**, como se muestra en la Figura 1-2. Puedes ingresar instrucciones de Python en la consola interactiva a a la derecha del prompt >>> y Python las ejecutará. Luego de mostrar los resultados de la instrucción, un nuevo prompt >>> estará esperando por tu próxima instrucción.

Figure 1-2: La consola interactiva del programa IDLE en Windows, OS X, y Ubuntu Linux.

## Cómo Usar este Libro

La mayoría de los capítulos en este libro comenzará con una muestra de ejecución del programa presentado en el capítulo en cuestión. Esta demostración revela cómo se ve el programa cuando lo ejecutas. El texto introducido por el usuario se **muestra en negrita**.

Teclea tú mismo el código del programa en el editor de archivos de IDLE, en lugar de descargarlo o copiarlo y pegarlo. Recordarás mejor cómo programar si te tomas el tiempo para escribir tú mismo el código.

### *Números de Línea y Espacios*

Al teclear el código de este libro, **no escribas** los números de línea que aparecen al principio de cada línea. Por ejemplo, si ves esto en el libro:

```
9. número = random.randint(1, 20)
```

o necesitas teclear el "9." a la izquierda, o el espacio a continuación. Sólo tecléalo así:

```
número = random.randint(1, 20)
```

Esos números están ahí sólo para que este libro pueda referir a líneas específicas del programa. No son parte del código fuente de un programa real.

Aparte de los números de línea, escribe el código exactamente como aparece. Ten en cuenta que algunas de las líneas de código están indentadas por cuatro u ocho espacios. Cada caracter en IDLE ocupa el mismo ancho, de modo que puedes contar el número de espacios contando el número de caracteres en las líneas arriba o abajo.

Por ejemplo, los espacios indentados aquí están marcados con un ▪ cuadrado negro para que puedas verlos:

```
while intentos < 10:
▪▪▪▪if número == 42:
▪▪▪▪▪▪▪▪print('Hola')
```

### *Ajuste de Texto en Este Libro*

Algunas instrucciones son demasiado largas para caber en una línea de la página por lo que continuarán en la línea siguiente. Al tipear este código, escríbelo todo en una línea sin pulsar INTRO. Puedes darte cuenta cuándo comienza una nueva instrucción mirando los números de línea a la izquierda del código. El ejemplo mostrado a continuación contiene sólo dos líneas:

```
1. print(¡Esta es la primera instrucción! xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxx')
2. print('Esta es la segunda instrucción, no la tercera.')
```

La primera instrucción continúa en el siguiente renglón y da el aspecto de que hubiera tres instrucciones en total. Esto es sólo porque las páginas de este libro no son lo suficientemente anchas para contener la primera instrucción en una sola línea de texto.

## Buscando Ayuda Online

El sitio web de este libro es http://inventwithpython.com/es. Puedes encontrar varios recursos relacionados con este libro allí. Varios enlaces de este libro utilizan el dominio invpy.com para direcciones URL acortadas.

El sitio web en http://reddit.com/r/inventwithpython es un lugar estupendo para para hacer preguntas de programación relacionadas con este libro. Publica preguntas generales sobre Python en los sitios web LearnProgramming y LearnPython en http://reddit.com/r/learnprogramming y http://reddit.com/r/learnpython respectivamente. http://translate.google.com puede realizar la traducción de Inglés para usted.

Asimismo, envíame por correo electrónico tus preguntas de programación a *al@inventwithpython.com*.

Ten presente que hay formas inteligentes para hacer preguntas de programación que ayudan otros a ayudarte. Asegúrate de leer las secciones de Preguntas Frecuentes que estos sitios web tienen acerca de la forma correcta de publicar preguntas. Cuando hagas preguntas de programación, haz lo siguiente:

- Si estás escribiendo a mano los programas de este libro y obtienes un error, primero busca errores tipográficos con la herramienta diff en http://invpy.com/es/diff. Copia y

pega tu código en la herramienta diff para encontrar las diferencias entre el código del libro y tu programa.
- Explica lo que estás intentando hacer cuando expliques el error. Esto permitirá a quien te ayuda saber si estás equivocándote por completo.
- Copia y pega el mensaje de error completo y tu código.
- Busca en la web para ver si alguien ya ha formulado (y respondido) tu pregunta.
- Explica lo que ya has intentado hacer para resolver tu problema. Esto muestra a la gente que ya has hecho algo de trabajo para tratar de entender las cosas por tí mismo.
- Sé amable. No exijas ayuda o presiones a quienes te ayudan para que respondan rápido.

Preguntar a alguien, "¿Por qué no está funcionando mi programa?" no le brinda ninguna información. Comunica a la persona qué es lo que estás intentando hacer, exactamente qué mensaje de error obtienes y qué versión de sistema operativo estás usando.

## *Resumen*

Este capítulo te ha ayudado a comenzar con el software Python mostrándote el sitio web http://python.org, de donde puedes descargarlo gratis. Luego de instalar y lanzar el software Python IDLE, estarás listo para aprender a programar a comenzando en el próximo capítulo.

El sitio web de este libro en http://inventwithpython.com/es contiene más información sobre cada uno de los capítulos, incluyendo un sitio web de trazado en línea y una herramienta diff que puede ayudarte a entender los programas de este libro.

# Capítulo 2
# LA CONSOLA INTERACTIVA

Temas Tratados En Este Capítulo:
- Enteros y Números de Punto Flotante
- Expresiones
- Valores
- Operadores
- Evaluación de Expresiones
- Almacenamiento de Valores en Variables

Antes de poder crear juegos, necesitas aprender algunos conceptos básicos de programación. No crearás juegos en este capítulo, pero aprender estos conceptos es el primer paso para programar videojuegos. Comenzaremos por aprender cómo usar la consola interactiva de Python.

## Operaciones Matemáticas Sencillas

Abre IDLE usando los pasos en el Capítulo 1, y haz que Python resuelva algunas operaciones matemáticas sencillas. La consola interactiva puede funcionar como una calculadora. Escribe `2 + 2` en la consola interactiva y presiona la tecla INTRO en tu teclado. (En algunos teclados, esta tecla se llama RETURN.) La Figura 2-1 muestra cómo IDLE responde con el número 4.

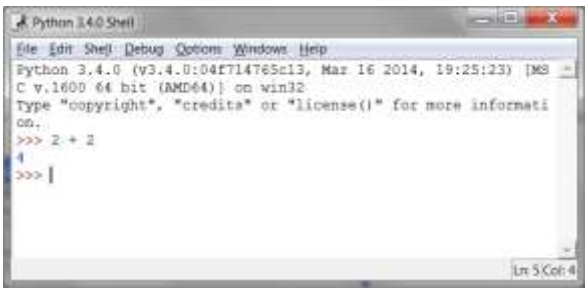



Figura 2-1: Escribe 2+2 en la consola interactiva.

Este problema matemático es una simple instrucción de programación. El signo + le dice a la computadora que sume los números 2 y 2. La Tabla 2-1 presenta los otros operadores matemáticos disponibles en Python. El signo - restará números. El asterisco * los multiplicará. La barra / los dividirá.

Tabla 2-1: Los diferentes operadores matemáticos en Python.

| Operador | Operación |
|---|---|
| + | suma |
| - | resta |
| * | multiplicación |
| / | división |

Cuando se usan de esta forma, +, -, *, y / se llama **operadores**. Los operadores le dicen a Python qué hacer con los números que los rodean.

## Enteros y Números de Punto Flotante

**Los enteros** (o **ints** para abreviar) son precisamente números enteros como 4, 99, y 0. **Los números de punto flotante** (o **floats** para abreviar) son fracciones o números con punto decimal como 3.5, 42.1 y 5.0. En Python, el número 5 is an integer, pero 5.0 es un float. A estos números se los llama **valores**.

## Expresiones

Estos problemas matemáticos son ejemplos de expresiones. Las computadoras pueden resolver millones de estos problemas en segundos. **Las expresiones** se componen de valores (los números) conectadas por operadores (los símbolos matemáticos). Prueba escribir algunos de estos problemas matemáticos en la consola interactiva, presiona la tecla INTRO después de cada uno.

```
2+2+2+2+2
8*6
10-5+6
2   +         2
```

Luego de introducir estas instrucciones, la consola interactiva se verá como la Figura 2-2.

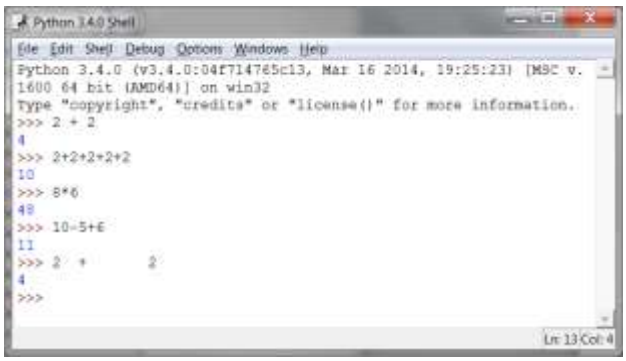



Figura 2-2: Así se ve la ventana de IDLE luego de introducir las instrucciones.


Figura 2-3: Una expresión se compone de valores y operadores.

En el ejemplo `2  +  2`, se ve que puede haber cualquier cantidad de espacios entre los valores y los operadores. Pero cada instrucción que escribas en la consola interactiva debe comenzar una línea.

## Evaluación de Expresiones

Cuando una computadora resuelve la expresión 10 + 5 y obtiene el valor 15, ha **evaluado** la expresión. Evaluar una expresión *la reduce a un único valor*, igual que resolver un problema de matemática lo reduce a un único número: la respuesta. Ambas expresiones `10 + 5` y `10 + 3 + 2` son evaluadas a `15`.

Las expresiones pueden ser de cualquier tamaño, pero siempre serán evaluadas a un valor único. Incluso valores únicos son expresiones: La expresión `15` se evalúa al valor `15`. Por ejemplo, la expresión `8 * 3 / 2 + 2 + 7 - 9` se evalúa al valor `12.0` a través de los siguientes pasos:

```
8 * 3 / 2 + 2 + 7 - 9
                  ▼
   24 / 2 + 2 + 7 - 9
                  ▼
     12.0 + 2 + 7 - 9
                  ▼
         14.0 + 7 - 9
                  ▼
             21.0 - 9
                  ▼
                 12.0
```

No puedes ver todos estos pasos en la consola interactiva. La consola los realiza y sólo te muestra los resultados:

```
>>> 8 * 3 / 2 + 2 + 7 - 9
12.0
```

Observa que el operador división `/` se evalúa a un valor float, como ocurre cuando `24 / 2` devuelve `12.0`. Las operaciones matemáticas con valores flotantes también devuelven valores flotantes, como cuando `12.0 + 2` devuelve `14.0`.

Notice that the `/` division operator evaluates to a float value, as in `24 / 2` evaluating to `12.0`. Math operations with float values also evaluate to float values, as in `12.0 + 2` evaluating to `14.0`.

### *Errores de Sintaxis*

Si escribes `5 +` en la consola interactiva, obtendrás un mensaje de error.

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

Este ocurre porque `5 +` no es una expresión. Las expresiones conectan valores mediante operadores. Pero el operador + espera un valor después del signo +. Cuando este valor no se encuentra, aparece un mensaje de error.

`SyntaxError` significa que Python no entiende la instrucción porque la has escrito de forma incorrecta. Una gran parte de programar computadoras se trata no sólo de decirle a la computadora qué hacer, sino también de saber cómo decírselo.

Pero no te preocupes por cometer errores. Los errores no dañan tu computadora. Simplemente vuelve a escribir la instrucción correctamente en la consola interactiva luego del siguiente indicador >>> de consola.

## Almacenamiento de Valores en Variables

Puedes guardar el valor al cual al cual una expresión es evaluada para poder usarlo más adelante en el programa, almacenándolo en una **variable**. Piensa una variable como una caja capaz de contener un valor.

Una **instrucción de asignación** guardará un valor dentro de una variable. Escribe el nombre de una variable seguido por el signo = (llamado **operador de asignación**), y luego el valor a almacenar en la variable. Por ejemplo, ingresa `spam = 15` en la consola interactiva:

```
>>> spam = 15
>>>
```

La caja de la variable `spam` tendrá guardado el valor `15`, como se muestra en la Figura 2-4. El nombre "spam" es la etiqueta en la caja (para que Python pueda distinguir las variables) y el valor está escrito en una pequeña nota dentro de la caja.

Cuando presiones INTRO no recibirás ninguna respuesta. En Python, si no aparece ningún mensaje de error significa que la instrucción se ha ejecutado correctamente. El indicador de consola >>> aparecerá para que puedas tipear la próxima instrucción.


Figura 2-4: Las variables son como cajas que pueden contener valores.

A diferencia de las expresiones, **las sentencias** no son evaluadas a ningún valor. Es por eso que no se muestra ningún valor en la siguiente línea de la consola interactiva a continuación de `spam = 15`. Puede ser confuso diferenciar cuáles instrucciones son expresiones y cuáles son sentencias. Sólo recuerda que las expresiones son evaluadas a un valor único. Cualquier otro tipo de instrucción es una sentencia.

Las variables almacenan valores, no expresiones. Por ejemplo, considera la expresión en las sentencias `spam = 10 + 5` y `spam = 10 + 7 - 2`. Ambas son evaluadas a `15`. El resultado final es el mismo: Las dos sentencias de asignación almacenan el valor `15` en la variables `spam`.

La primera vez que una variables es usada en una sentencia de asignación, Python creará esa variable. Para comprobar qué valor contiene una variable dada, escribe el nombre de la variable en la consola interactiva:

```
>>> spam = 15
>>> spam
15
```

The expression `spam` evaluates to the value inside the `spam` variable: `15`. You can use variables in expressions. Try entering the following in the interactive shell:

```
>>> spam = 15
>>> spam + 5
20
```

Haz fijado el valor de la variable spam en 15, por lo que escribir `spam + 5` es como escribir la expresión `15 + 5`. Aquí se muestran los pasos para la evaluación de `spam + 5`:

```
spam + 5
     ▼
 15  + 5
     ▼
    20
```

No puedes usar una variable antes de que sea creada por una sentencia de asignación. Python responderá con `NameError` porque todavía no existe una variable con ese nombre. Escribir mal el nombre de una variable también causa este error:

```
>>> spam = 15
>>> spma
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    spma
NameError: name 'spma' is not defined
```

El error aparece porque hay una variable llamada `spam`, pero ninguna llamada `spma`.

Puedes cambiar el valor almacenado en una variable escribiendo otra sentencia de asignación. Por ejemplo, prueba escribir lo siguiente en la consola interactiva:

```
>>> spam = 15
>>> spam + 5
20
>>> spam = 3
>>> spam + 5
8
```

La primera vez que escribes `spam + 5`, la expresión se evalúa a `20` porque has guardado `15` dentro de `spam`. Sin embargo, cuando escribes `spam = 3`, el valor `15` es reemplazado, o **sobrescrito**, con el valor `3`. Ahora cuando escribes `spam + 5`, la expresión se evalúa a `8` porque el valor de `spam` es ahora `3`. La sobrescritura se muestra en la Figura 2-5.


Figura 2-5: El valor 15 en spam es sobrescrito por el valor 3.

Puedes incluso usar el valor en la variable `spam` para asignar un nuevo valor a `spam`:

```
>>> spam = 15
>>> spam = spam + 5
20
```

La sentencia de asignación `spam = spam + 5` es como decir, "el nuevo valor de la variable spam será el valor actual de `spam` más cinco". Continúa incrementando el valor de `spam` en `5` varias veces escribiendo lo siguiente en la consola interactiva:

```
>>> spam = 15
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam = spam + 5
>>> spam
30
```

## *Usando Más De Una Variable*

Crea tantas variables como necesites en tus programas. Por ejemplo, asignemos diferentes valores a dos variables llamadas `eggs` y `bacon`, de esta forma:

```
>>> bacon = 10
>>> eggs = 15
```

Ahora la variable `bacon` almacena el valor `10`, y `eggs` almacena el valor `15`. Cada variable es una caja independiente con su propio valor, como en la Figura 2-6.

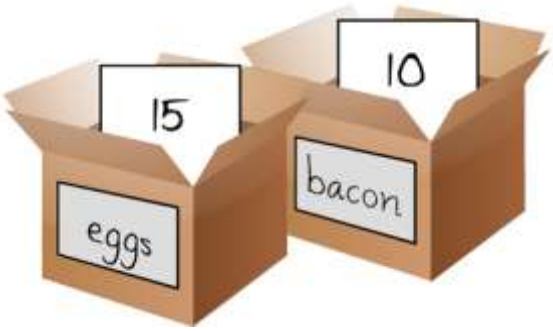



Figura 2-6: Las variables “bacon” y “eggs” almacenan valores dentro de ellas.

Intenta escribir `spam = bacon + eggs` en la consola interactiva, luego comprueba el nuevo valor de `spam`:

```
>>> bacon = 10
>>> eggs = 15
>>> spam = bacon + eggs
>>> spam
25
```

El valor de `spam` es ahora `25`. Cuando sumas `bacon` y `eggs` estás sumando sus valores, que son `10` y `15` respectivamente. Las variables contienen valores, no expresiones. La variable `spam` recibió el valor `25`, y no la expresión `bacon + eggs`. Luego de la sentencia de asignación `spam = bacon + eggs`, cambiar `bacon` o `eggs` no afecta a `spam`.

## *Resumen*

n este capítulo has aprendido los conceptos básicos para escribir instrucciones en Python. Python necesita que le digas exactamente qué hacer de forma estricta. Las computadoras no tienen sentido común y sólo entienden instrucciones específicas.

Las expresiones son valores (tales como `2` ó `5`) combinados con operadores (tales como `+` o `-`). Python puede evaluar expresiones, es decir, reducirlas a un valor único. Puedes almacenar valores dentro de las variables de modo que tu programa sea capaz de recordarlas y usarlas más adelante.

Hay muchos otros tipos de operadores y valores en Python. En el próximo capítulo, repasaras algunos conceptos más y escribirás tu primer programa. Aprenderás a trabajar con texto en expresiones. Python no está limitado a números; ¡es más que sólo una calculadora!

# Capítulo 3
# ESCRIBIENDO PROGRAMAS

Temas Tratados En Este Capítulo:
- Flujo de ejecución
- Cadenas
- Concatenación de cadenas
- Tipos de datos (como cadenas o enteros)
- Usando el editor de archivos para escribir programas
- Guardar y ejecutar programas en IDLE
- La función print()
- La función input()
- Comentarios
- Sensibilidad a mayúsculas

Suficiente matemática por ahora. Ahora veamos qué puede hacer Python con texto. En este capítulo, aprenderás cómo almacenar texto en variables, combinar textos, y mostrar texto en pantalla.

Casi todos los programas muestran texto al usuario, y el usuario ingresa texto en tus programas a través del teclado. En este capítulo crearás tu primer programa. Este programa muestra el saludo "¡Hola Mundo!" y te pregunta tu nombre.

## Cadenas

En Python, los valores de texto se llaman **cadenas**. Los valores cadena pueden usarse igual que valores enteros o float. Puedes almacenar cadenas en variables. En código, las cadenas comienzan y terminan con una comilla simple (`'`). Prueba introducir este código en la consola interactiva:

```
>>> spam = 'hola'
```

Las comillas simples le dicen a Python dónde comienza y termina la cadena, pero no son parte del texto del valor de cadena. Ahora bien, si escribes `spam` en la consola interactiva, podrás ver el contenido de la variable `spam`. Recuerda, Python evalúa las variables al valor almacenado dentro de las mismas. En este caso, la cadena `'hola'`:

```
>>> spam = 'hola'
>>> spam
```

```
'hola'
```

Las cadenas pueden contener cualquier caracter del teclado y pueden ser tan largas como quieras. Todos estos son ejemplos de cadenas:

```
'hola'
'¡Oye tú!'
'GATITOS'
'7 manzanas, 14 naranjas, 3 limones'
'Si no está relacionado con elefantes es irrelefante.'
'Hace mucho tiempo, en una galaxia muy, muy lejana...'
'O*&#wY%*&OCfsdYO*&gfC%YO*&%3yc8r2'
```

## Concatenación de cadenas

Las cadenas pueden combinarse con operadores para generar expresiones, al igual que los números enteros y floats. Puedes combinar dos cadenas con el operador `+`. Esto es **concatenación de cadenas**. Prueba ingresando `'¡Hola' + 'Mundo!'` into the interactive shell:

```
>>> '¡Hola' + 'Mundo!'
'¡HolaMundo!'
```

La expresión se evalúa a un valor único de cadena, `'¡HolaMundo!'`. No hay un espacio entre las palabras porque no había espacios en ninguna de las cadenas concatenadas, a diferencia del siguiente ejemplo:

```
>>> '¡Hola ' + 'Mundo!'
'¡Hola Mundo!'
```

El operador `+` funciona de forma diferente sobre valores enteros y cadenas, ya que son distintos tipos de datos. Todos los valores tienen un tipo de datos. El tipo de datos del valor `'Hola'` es una cadena. El tipo de datos del valor `5` es un entero. El tipo de datos le dice a Python qué deben hacer los operadores al evaluar expresiones. El operador `+` concatena valores de tipo cadena, pero suma valores de tipo entero (o float).

## Escribir Programas en el Editor de Archivos de IDLE

Hasta ahora has estado escribiendo instrucciones, una a la vez, en la consola interactiva de IDLE. Cuando escribes programas, sin embargo, escribes varias instrucciones y haces que se ejecuten a la vez. ¡Escribamos ahora tu primer programa!

IDLE tiene otra parte llamada **el editor de archivos**. Haz clic en el menú **File** (Archivo) en la parte superior de la ventana de la consola interactiva. Luego selecciona **New Window** (Nueva

Ventana). Aparecerá una ventana vacía para que escribas el código de tu programa, como se ve en la Figura 3-1.

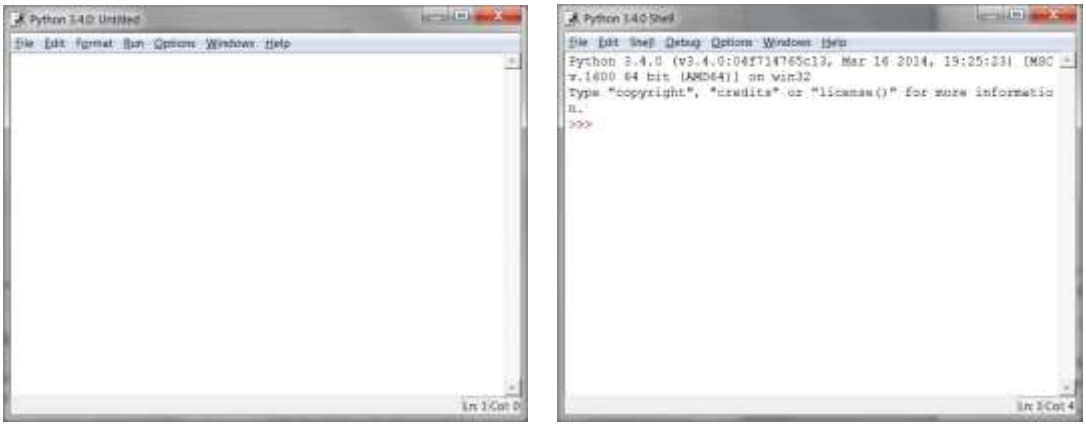


Figura 3-1: La ventana del editor de archivos (izquierda) y la consola interactiva (derecha).

Las dos ventanas se ven parecidas, pero sólo recuerda esto: **La ventana de la consola interactiva** tendrá el símbolo de sistema >>>. **La ventana del editor de archivos** no lo tendrá.

## ¡Hola Mundo!

Es tradición entre programadores hacer que su primer programa muestre "¡Hola Mundo!" en la pantalla. Ahora crearás tu propio programa Hola Mundo.

Al ingresar tu programa, no escribas los números a la izquierda del código. Están allí sólo para que este libro pueda referirse al código por número de línea. La esquina inferior derecha de la ventana del editor de archivos te indicará dónde está el cursor intermitente. La Figura 3-2 muestra que el cursor se encuentra sobre la línea 1 y sobre la columna 0.

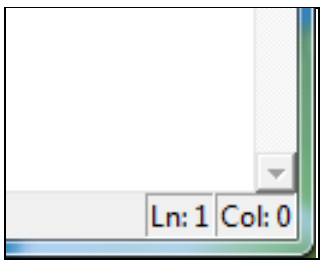



Figura 3-2: La parte inferior derecha de la ventana del editor de archivos te indica en qué línea está el cursor.

### *hola.py*

Ingresa el siguiente texto en la nueva ventana del editor de archivos. Este es el **código fuente del programa**. Contiene las instrucciones que Python seguirá cuando el programa se ejecute.

> **¡NOTA IMPORTANTE!** Los programas de este libro sólo podrán ejecutarse sobre Python 3, no Python 2. Al iniciar la ventana IDLE, dirá algo como "Python 3.4.2" en la parte superior. Si tienes Python 2 instalado, es posible instalar también Python 3 a la vez. Para descargar Python 3, dirígete a https://python.org/download/.

hola.py

```
1. # Este programa saluda y pregunta por mi nombre.
2. print('¡Hola mundo!')
3. print('¿Cómo te llamas?')
4. miNombre = input()
5. print('Es un placer conocerte, ' + miNombre)
```

El programa IDLE escribirá diferentes tipos de instrucciones en diferentes colores. Cuando hayas terminado de escribir el código, la ventana debería verse así:

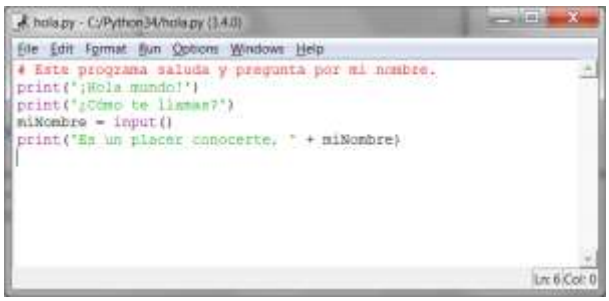



Figura 3-3: La ventana del editor de archivos se verá así luego de haber ingresado el código.

## Guardando el programa.

Una vez que hayas ingresado tu código fuente, guárdalo haciendo clic en **File** (Archivo) ► **Save As** (Guardar Como). O pulsa Ctrl-S para guardar usando un acceso directo del teclado. La Figura 3-4 muestra la ventana Guardar Como que se abrirá. Escribe *hola.py* en el campo de texto **Nombre** y haz clic en **Guardar**.

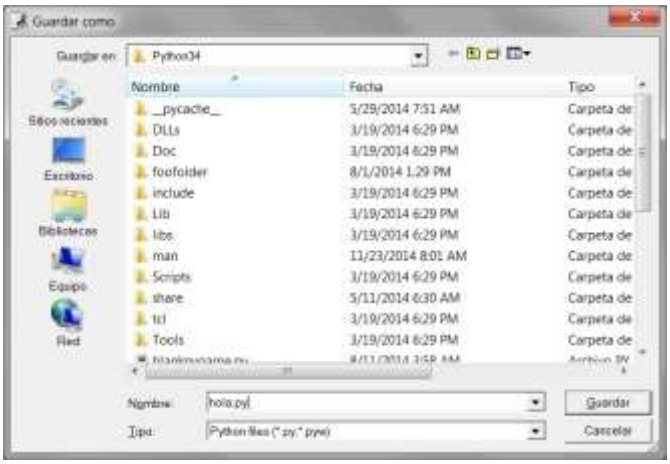



Figura 3-4: Guardando el programa.

Deberías guardar tus programas a menudo. De esta manera, si el ordenador se bloquea o accidentalmente sales de IDLE no perderás mucho trabajo.

## Abriendo tus Programas Guardados

Para cargar un programa guardado, haz clic en **File** (Archivo) ► **Open** (Abrir). Elige el archivo en la ventana que aparece y haz clic en el botón **Open** (Abrir). Tu programa *hola.py* se abrirá en la ventana del Editor de Archivos.

Es hora de ejecutar el programa. Haz clic en **File** (Archivo) ► **Run** (Ejecutar) ► **Run Module** (Ejecutar Módulo) o simplemente pulsa F5 desde la ventana del editor de archivos. Tu programa se ejecutará en la ventana de la consola interactiva.

Escribe tu nombre cuando el programa lo pida. Esto se verá como en la Figura 3-5:

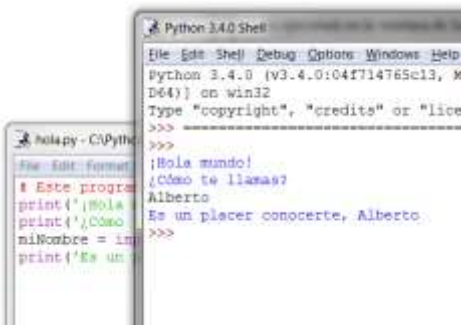



Figura 3-5: La consola interactiva luego de ejecutar hola.py.

Cuando escribas tu nombre y pulses INTRO, el programa te saludará por tu nombre. ¡Felicitaciones! Haz escrito tu primer programa y ya eres un programador. Pulsa F5 de nuevo para volver a ejecutar el programa y esta vez escribe otro nombre.

Si has obtenido un error, compara tu código con el de este libro usando la herramienta online diff en http://invpy.com/es/diff/hola. Copia y pega tu código del editor de archivos en la página web y haz clic en el botón **Comparar**. Esta herramienta resaltará cualquier diferencia entre tu código y el código en este libro, como en la Figura 3-6.

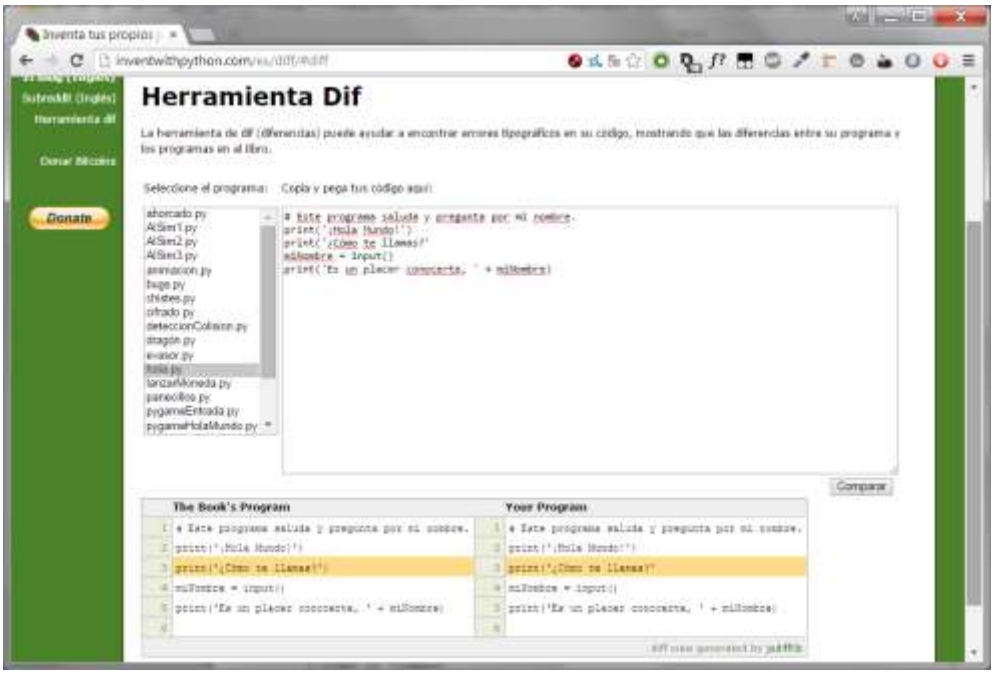



Figure 3-6: La herramienta diff en http://invpy.com/es/diff.

While coding, if you get a `NameError` that looks like this:

```
¡Hola mundo!
¿Cómo te llamas?
Alberto
Traceback (most recent call last):
  File "C:/Python26/test1.py", line 4, in <module>
    miNombre = input()
  File "<string>", line 1, in <module>
NameError: name 'Alberto' is not defined
```

...quiere decir que estás usando Python 2, en lugar de Python 3. Instala una versión de Python 3 de http://python.org. Luego, re-ejecuta el programa con Python 3.

## Cómo Funciona el Programa "Hola Mundo"

Cada línea de código es una instrucción interpretada por Python. Estas instrucciones constituyen el programa. Las instrucciones de un programa de computadora son como los pasos en una receta de un libro de cocina. Cada instrucción se ejecuta en orden, comenzando por la parte superior del programa y en dirección descendente hasta el final de la lista de instrucciones.

El paso del programa en el cual Python se encuentra se llama **ejecución**. Cuando el programa comienza, la ejecución se encuentra en la primera instrucción. Luego de ejecutarla, la ejecución baja hasta la próxima instrucción.

Veamos cada línea de código para entender qué es lo que hace. Comenzaremos por la línea número 1.

### *Comentarios*

```
1. # Este programa saluda y pregunta por mi nombre.
```

Esta instrucción es un **comentario**. Cualquier texto a continuación del signo # (llamado **símbolo almohadilla**) es un comentario. Los comentarios no son para Python, sino para tí, el programador. Python ignora los comentarios. Los comentarios son notas del programador acerca de lo que el código hace. Puedes escribir lo que quieras en un comentario. Para facilitar la lectura del código fuente, este libro muestra los comentarios en texto de color gris claro.

Los programadores usualmente colocan un comentario en la parte superior de su código para dar un título a su programa.

### *Funciones*

Una **función** es una especie de mini-programa dentro de tu programa. Las funciones contienen instrucciones que se ejecutan cuando la función es llamada. Python ya tiene algunas funciones integradas. Dos funciones, `print()` e `input()`, son descriptas a continuación. Lo maravilloso acerca de las funciones es que sólo necesitas saber lo que la función hace, pero no cómo lo hace.

Una **llamada a una función** es un fragmento de código que dice a Python que ejecute el código dentro de una función. Por ejemplo, tu programa llama a la función `print()` para mostrar una cadena en la pantalla. La función `print()` toma la cadena que tú escribes entre los paréntesis como entrada y muestra el texto en la pantalla.

Para mostrar `¡Hola mundo!` en la pantalla, escribe el nombre de la función print, seguido por un paréntesis de apertura, seguido por la cadena `'¡Hola mundo!'` y un paréntesis de cierre.

## La función `print()`

```
2. print('¡Hola mundo!')
3. print('¿Cómo te llamas?')
```

Las líneas 2 y 3 son llamadas a la función `print()`. Un valor entre los paréntesis de la llamada a una función es un **argumento**. El argumento en la llamada a la función `print()` de la línea 2 es `'¡Hola mundo!'`. El argumento en la llamada a `print()` de la línea 3 es `'¿Cómo te llamas?'`. Esto se llama **pasar** el argumento a la función `print()`.

En este libro, los nombres de funciones tienen paréntesis al final. Esto deja en claro que `print()` hace referencia a una función llamada `print()`, y no a una variable llamada `print`. Esto es como el uso de comillas alrededor del número `'42'` para indicar a Python that que estás refiriéndote a la cadena `'42'` y no al entero `42`.

## La función `input()`

```
4. miNombre = input()
```

La línea 4 es una sentencia de asignación con una variable (`miNombre`) y una llamada a una función (`input()`). Cuando `input()` es llamada, el programa espera a que el usuario ingrese texto. La cadena de texto que el usuario ingresa se convierte en el valor al que se evalúa la llamada a la función. Las llamadas a funciones pueden usarse en expresiones, en cualquier lugar en que pueda usarse un valor.

El valor al cual se evalúa la llamada a la función es llamado **valor de retorno**. (De hecho, "el valor devuelto por la llamada a una función" significa lo mismo que "el valor al que se evalúa la llamada a la función".) En este caso, el valor devuelto por la función `input()` es la cadena que el usuario ha escrito (su nombre). Si el usuario ha ingresado "Alberto", la llamada a la función `input()` se evalúa a la cadena `'Alberto'`. La evaluación se ve así:

```
miNombre = input()
           ▼
miNombre = 'Alberto'
```

Así es como el valor de cadena `'Alberto'` es almacenado en la variable `miNombre`.

## Uso de Expresiones en Llamadas a Funciones

```
5. print('Es un placer conocerte, ' + miNombre)
```

La última línea es otra llamada a la función `print()`. La expresión `'Es un placer conocerte, ' + miNombre` entre los paréntesis de `print()`. Sin embargo, los argumentos son siempre valores individuales. Python evaluará primero esta expresión y luego pasará este valor como argumento. Si `'Alberto'` está almacenado en `miNombre`, la evaluación ocurre así:

```
print(Es un placer conocerte, ' + miNombre)
                          ▼
print(Es un placer conocerte, ' + 'Alberto')
                          ▼
print(Es un placer conocerte, Alberto')
```

Así es como el programa saluda al usuario por su nombre.

### *Terminando el Programa*

Una vez que el programa ejecuta la última línea, **termina** y se **sale** del programa. Esto quiere decir que el programa deja de ejecutarse. Python olvida todos los valores almacenados en variables, incluyendo la cadena almacenada en `miNombre`. Si ejecutas el programa de nuevo y escribes un nombre diferente, el programa pensará que esa otra cadena es tu nombre.

```
¡Hola mundo!
¿Cómo te llamas?
Carolyn
Es un placer conocerte, Carolyn
```

Recuerda, la computadora hace exactamente lo que la programas para hacer. Las computadoras son tontas y sólo siguen tus instrucciones al pie de la letra. A la computadora no le importa si escribes tu nombre, el nombre de otra persona, o sólo algo absurdo. Escribe lo que quieras. La computadora lo tratará de la misma forma:

```
Hello world!
What is your name?
popó
Es un placer conocerte, popó
```

## Nombres de Variables

Dar nombres descriptivos a las variables facilita entender qué es lo que hace un programa. Imagina si estuvieses mudándote a una nueva casa y hubieses colocado a cada una de tus cajas la etiqueta “Cosas”. ¡Eso no sería útil en lo absoluto!

En lugar de `miNombre`, podrías haber llamado a esta variable `abrahamLincoln` o `nOmBrE`. A Python no le importa. Ejecutará el programa de la misma forma.

Los nombres de variables son sensibles a mayúsculas. **Sensible a mayúsculas** significa que el mismo nombre de variable con diferente capitalización se considera una variable diferente. De modo que `spam`, `SPAM`, `Spam`, y `sPAM` son cuatro variables diferentes en Python. Cada una de ellas contiene su propio valor independiente. Es una mala idea tener variables con diferente capitalización en tu programa. En lugar de ello, usa nombres descriptivos para tus variables.

Los nombres de variables se escriben habitualmente en minúscula. Si hay más de una palabra en el nombre de la variable, escribe en mayúscula la primera letra de cada palabra después de la primera. Esto hace que tu código sea más legible. Por ejemplo, el nombre de variable `loQueHeDesayunadoEstaMañana` es mucho más fácil de leer que `loquehedesayunadoestamañana`. Esto es una **convención**: una forma opcional pero estándar de hacer las cosas en Python.

Es preferible usar nombres cortos antes que largos a las variables: `desayuno` o `comidaEstaMañana` son más fáciles de leer que `loQueHeDesayunadoEstaMañana`.

Los ejemplos en este libro de la consola interactiva usan nombres de variables como `spam`, `eggs`, `ham`, y `bacon`. Esto es porque los nombres de variables en estos ejemplos no importan. Sin embargo, todos los programas de este libro usan nombres descriptivos. Tus programas también deberían usar nombres de variables descriptivos.

## *Resumen*

Luego de haber aprendido acerca de cadenas y funciones, puedes empezar a crear programas que interactúan con usuarios. Esto es importante porque texto es la principal vía de comunicación entre el usuario y la computadora. El usuario ingresa texto a través el teclado mediante la función `input()`, y la computadora muestra texto en la pantalla usando la función `print()`.

Las cadenas son simplemente valores de un nuevo tipo de datos. Todos los valores tienen un tipo de datos, y hay muchos tipos de datos en Python. El operador + se usa para unir cadenas.

Las funciones se usan para llevar a cabo alguna instrucción complicada como parte de nuestro programa. Python tiene muchas funciones integradas acerca de las cuales aprenderás en este libro. Las llamadas a funciones pueden ser usadas en expresiones en cualquier lugar donde se usa un valor.

La instrucción de tu programa en que Python se encuentra se denomina ejecución. En el próximo capítulo, aprenderás más acerca de cómo hacer que la ejecución proceda de otras formas que simplemente en forma descendente a través del programa. Una vez que aprendas esto, podrás comenzar a crear juegos.

# Capítulo 4

# ADIVINA EL NÚMERO

Temas Tratados En Este Capítulo:

- Sentencias `import`
- Módulos
- Sentencias `while`
- Condiciones
- Bloques
- Booleanos
- Operadores de comparación
- La diferencia entre = y ==
- Sentencias `if`
- La palabra reservada `break`
- Las funciones `str()`, `int()` y `float()`
- La función `random.randint()`

En este capítulo crearás el juego "Adivina el Número". La computadora pensará un número aleatorio entre 1 y 20, y te pedirá que intentes adivinarlo. La computadora te dirá si cada intento es muy alto o muy bajo. Tú ganas si adivinas el número en seis intentos o menos.

Este es un buen juego para codificar ya que usa números aleatorios y bucles, y recibe entradas del usuario en un programa corto. Aprenderás cómo convertir valores a diferentes tipos de datos, y por qué es necesario hacer esto. Dado que este programa es un juego, nos referiremos al usuario como **el jugador**. Pero llamarlo "usuario" también sería correcto.

## Muestra de ejecución de "Adivina el Número"

Así es como el programa se muestra al jugador al ejecutarse. El texto que el jugador ingresa está en **negrita**.

```
¡Hola! ¿Cómo te llamas?
Alberto
Bueno, Alberto, estoy pensando en un número entre 1 y 20.
Intenta adivinar.
10
Tu estimación es muy alta.
Intenta adivinar.
2
```

```
Tu estimación es muy baja.
Intenta adivinar.
4
¡Buen trabajo, Albert! ¡Has adivinado mi número en 3 intentos!
```

## Código Fuente de Adivina el Número

Abre una nueva ventana del editor de archivos haciendo clic en **File** (Archivo) ► **New Window** (Nueva Ventana). En la ventana vacía que aparece, escribe el código fuente y guárdalo como *adivinaElNúmero.py*. Luego ejecuta el programa pulsando F5. Cuando escribas este código en el editor de archivos, asegúrate de prestar atención a la cantidad de espacios delante de algunas de las líneas. Algunas líneas están indentadas por cuatro u ocho espacios.

> **¡NOTA IMPORTANTE!** Los programas de este libro sólo podrán ejecutarse sobre Python 3, no Python 2. Al iniciar la ventana IDLE, dirá algo como "Python 3.4.2" en la parte superior. Si tienes Python 2 instalado, es posible instalar también Python 3 a la vez. Para descargar Python 3, dirígete a https://python.org/download/.

Si obtienes errores luego de copiar este código, compáralo con el código del libro usando la herramienta diff online en http://invpy.com/diff/adivinaElNúmero.

adivinaElNúmero.py

```
 1. # Este es el juego de adivinar el número.
 2. import random
 3.
 4. intentosRealizados = 0
 5.
 6. print('¡Hola! ¿Cómo te llamas?')
 7. miNombre = input()
 8.
 9. número = random.randint(1, 20)
10. print('Bueno, ' + miNombre + ', estoy pensando en un número entre 1 y 20.')
11.
12. while intentosRealizados < 6:
13.     print('Intenta adivinar.') # Hay cuatro espacios delante de print.
14.     estimación = input()
15.     estimación = int(estimación)
16.
17.     intentosRealizados = intentosRealizados + 1
18.
19.     if estimación < número:
20.         print('Tu estimación es muy baja.') # Hay ocho espacios delante de
print.
```

```
21.
22.     if estimación > número:
23.         print('Tu estimación es muy alta.')
24.
25.     if estimación == número:
26.         break
27.
28. if estimación == número:
29.     intentosRealizados = str(intentosRealizados)
30.     print('¡Buen trabajo, ' + miNombre + '! ¡Has adivinado mi número en ' +
intentosRealizados + ' intentos!')
31.
32. if estimación != número:
33.     número = str(número)
34.     print('Pues no. El número que estaba pensando era ' + número)
```

## Sentencias `import`

```
1. # Este es el juego de adivinar el número.
2. import random
```

La primera línea es un comentario. Recuerda que Python ignorará todo lo que esté precedido por el signo `#`. Esto sólo nos indica qué es lo que hace el programa.

La segunda línea es una sentencia **import**. Recuerda, las sentencias son instrucciones que realizan alguna acción, pero no son evaluadas a un valor como las expresiones. Ya has visto sentencias antes: las sentencias de asignación almacenan un valor en una variable.

Aunque Python incluye muchas funciones integradas, algunas funciones existen en programas separados llamados **módulos**. Puedes usar estas funciones importando sus módulos en tu programa con una sentencia `import`.

La línea 2 importa el módulo llamado `random` de modo que el programa pueda llamar a `random.randint()`. Esta función generará un número aleatorio para que el usuario adivine.

```
4. intentosRealizados = 0
```

La línea 4 crea una nueva variable llamada `intentosRealizados`. Guardaremos en esta variable el número de veces que el jugador ha intentado adivinar el número. Ya que el jugador no ha realizado ningún intento a esta altura del programa, guardaremos aquí el entero `0`.

```
6. print('¡Hola! ¿Cómo te llamas?')
```

```
7. miNombre = input()
```

Las líneas 6 y 7 son iguales a las líneas en el programa Hola Mundo que viste en el Capítulo 3. Los programadores a menudo reutilizan código de sus otros programas para ahorrarse trabajo.

La línea 6 es una llamada a la función `print()`. Recuerda que una función es como un mini-programa dentro de tu programa. Cuando tu programa llama a una función, ejecuta este mini-programa. El código dentro de la función `print()` muestra en la pantalla la cadena que ha recibido como argumento.

La línea 7 permite al usuario escribir su nombre y lo almacena en la variable `miNombre`. (Recuerda, la cadena podría no ser realmente el nombre del jugador. Es simplemente cualquier cadena que el jugador haya introducido. Las computadoras son tontas, y sólo siguen sus instrucciones sin importarles nada más.)

## La Función `random.randint()`

```
9. número = random.randint(1, 20)
```

La línea 9 llama a una nueva función denominada `randint()` y guarda el valor que ésta devuelve en la variable `número`. Recuerda, las llamadas a funciones pueden ser parte de expresiones, ya que son evaluadas a un valor.

La función `randint()` es parte del módulo random, por lo que debes colocar `random.` delante de ella (¡no olvides colocar el punto!) para decirle a Python que la función `randint()` está en el módulo random.

La función `randint()` devolverá un entero aleatorio en el intervalo comprendido (incluidos los bordes) entre los dos argumentos enteros que le pases. La línea 9 pasa `1` y `20` separados por una coma y entre los paréntesis que siguen al nombre de la función. El entero aleatorio devuelto por `randint()` es almacenado en una variable llamada `número`; este es el número secreto que el jugador intentará adivinar.

Sólo por un momento, vuelve a la consola interactiva y escribe `import random` para importar el módulo `random`. Luego escribe `random.randint(1, 20)` para ver a qué se evalúa la llamada a la función. Devolverá un entero entre `1` y `20`. Repite el código nuevamente y la llamada a la función probablemente devolverá un entero diferente. La función `randint()` devuelve un entero aleatorio cada vez, de la misma forma en que tirando un dado obtendrías un número aleatorio cada vez:

```
>>> import random
>>> random.randint(1, 20)
12
```

```
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
3
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
7
```

Usa la función `randint()` cuando quieras agregar aleatoriedad a tus juegos. Y vas a usar aleatoriedad en muchos juegos. (Piensa en la cantidad de juegos de mesa que utilizan dados.)

También puedes probar diferentes intervalos de números cambiando los argumentos. Por ejemplo, escribe `random.randint(1, 4)` para obtener sólo enteros entre `1` y `4` (incluyendo `1` y `4`). O prueba `random.randint(1000, 2000)` para obtener enteros entre `1000` y `2000`.

Por ejemplo, escribe lo siguiente en la consola interactiva. Los resultados que obtienes cuando llamas a la función `random.randint()` serán seguramente diferentes (después de todo es aleatorio).

```
>>> random.randint(1, 4)
3
>>> random.randint(1000, 2000)
1294
```

Puedes cambiar ligeramente el código fuente del juego para hacer que el programa se comporte de forma diferente. Prueba cambiar las líneas 9 y 10 de:

```
 9. número = random.randint(1, 20)
10. print('Bueno, ' + miNombre + ', estoy pensando en un número entre 1 y 20.')
```

…a lo siguiente:

```
 9. número = random.randint(1, 100)
10. print('Bueno, ' + miNombre + ', estoy pensando en un número entre 1 y
100.')
```

Y ahora la computadora pensará en un entero comprendido entre `1` y `100` en lugar de entre `1` y `20`. Cambiar la línea 9 cambiará el intervalo del número aleatorio, pero recuerda cambiar también la línea 10 para que el juego le diga al jugador el nuevo rango en lugar del viejo.

### *Recibiendo al Jugador*

```
10. print('Bueno, ' + miNombre + ', estoy pensando en un número entre 1 y 20.')
```

En la línea 10 la función `print()` recibe al jugador llamándolo por su nombre, y le dice que la computadora está pensando un número aleatorio.

Puede parecer que hay más de un argumento cadena en la línea 10, pero observa la línea con cuidado. El signo suma concatena las tres cadenas de modo que son evaluadas a una única cadena. Y esa única cadena es el argumento que se pasa a la función `print()`. Si miras detenidamente, verás que las comas están dentro de las comillas, por lo que son parte de las cadenas y no un separador.

## Bucles

```
12. while intentosRealizados < 6:
```

La línea 12 es una sentencia `while` (mientras), que indica el comienzo de un bucle `while`. **Los bucles** te permiten ejecuta código una y otra vez. Sin embargo, necesitas aprender algunos otros conceptos antes de aprender acerca de los bucles. Estos conceptos son bloques, booleanos, operadores de comparación, condiciones, y la sentencia `while`.

## Bloques

Varias líneas de código pueden ser agrupadas en un bloque. Un **bloque** consiste en líneas de código que comparten mínima indentación posible. Puedes ver dónde comienza y termina un bloque de código mirando el número de espacios antes de las líneas. Esto se llama la **indentación** de la línea.

Un bloque comienza cuando la indentación de una línea se incrementa (usualmente en cuatro espacios). Cualquier línea subsiguiente que también esté indentada por cuatro espacios es parte del bloque. El bloque termina cuando hay una línea de código con la misma indentación que antes de empezar el bloque. Esto significa que pueden existir bloques dentro de otros bloques. La Figura 4-1 es un diagrama de código con los bloques delineados y numerados. Los espacios son cuadrados negros para que sean más fáciles de contar.

En la Figura 4-1, la línea 12 no tiene indentación y no se encuentra dentro de ningún bloque. La línea 13 tiene una indentación de cuatro espacios. Como esta indentación es mayor que la indentación de la línea anterior, ha comenzado un nuevo bloque. Este bloque tiene la etiqueta (1)

en la Figura 4-1. Este bloque continuará hasta una línea sin espacios (la indentación original antes de que comenzara el bloque). Las líneas vacías son ignoradas.

La línea 20 tiene una indentación de ocho espacios. Ocho espacios es más que cuatro espacios, lo que comienza un nuevo bloque. Este bloque se señala con (2) en la Figura 4-1. Este bloque se encuentra dentro de otro bloque.

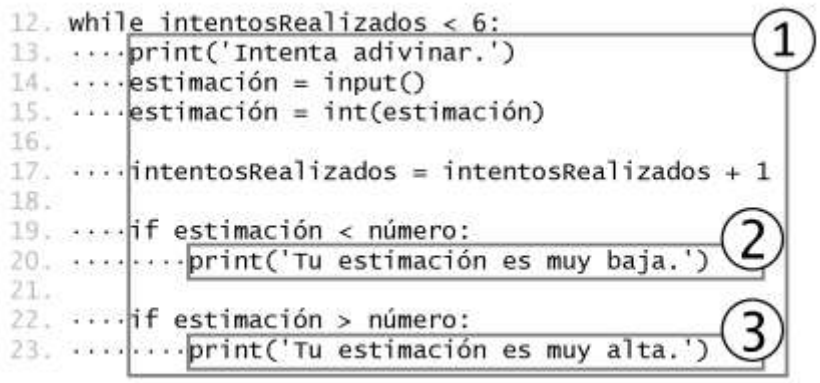



Figura 4-1: Bloques y su indentación. Los puntos negros representan espacios.

La línea 22 sólo tiene cuatro espacios. Al ver que la indentación se ha reducido, sabes que el bloque ha terminado. La línea 20 es la única línea del bloque. La línea 22 está en el mismo bloque que las otras líneas con cuatro espacios.

La línea 23 incrementa la indentación a ocho espacios, de modo que otra vez comienza un nuevo bloque. Es el que tiene la etiqueta (3) en la Figura 4-1.

Para recapitular, la línea 12 no están en ningún bloque. Las líneas 13 a 23 pertenecen al mismo bloque (marcado como bloque 1). La línea 20 está en un bloque dentro de un bloque marcado con (2). Y la línea 23 es la única línea en otro bloque dentro de un bloque marcado con (3).

## El Tipo de Datos Booleano

El tipo de datos Booleano tiene sólo dos valores: `True` (Verdadero) o `False` (Falso). Estos valores deben escribirse con “T” y “F” mayúsculas. El resto del nombre del valor debe estar en minúscula. Usarás valores Booleanos (llamados **bools** por brevedad) con operadores de comparación para formar condiciones. (Las condiciones serán explicadas más adelante.)

Por ejemplo:

```
>>> spam = True
```

```
>>> eggs = False
```

Los tipos de datos que han sido introducidos hasta ahora son enteros, floats, cadenas, y ahora bools.

## Operadores de Comparación

La línea 12 tiene una sentencia `while`:

```
12. while intentosRealizados < 6:
```

La expresión que sigue a la palabra reservada while (la parte `intentosRealizados < 6`) contiene dos valores (el valor en la variable `intentosRealizados`, y el valor entero `6`) conectados por un operador (el símbolo `<`, llamado el símbolo "menor que"). El símbolo `<` se llama un **operador de comparación**.

Los operadores de comparación comparan dos valores y se evalúan a un valor Booleano True o False. En la Tabla 4-1 se muestra una lista de todos los operadores de comparación.

Table 4-1: Operadores de comparación.

| Signo del Operador | Nombre del Operador |
|---|---|
| < | Menor que |
| > | Mayor que |
| <= | Menor o igual a |
| >= | Mayor o igual a |
| == | Igual a |
| != | Diferente a |

Ya has leído acerca de los operadores matemáticos `+`, `-`, `*`, y `/`. Como cualquier operador, los operadores de comparación se combinan con valores ara formar expresiones tales como `intentosRealizados < 6`.

## Condiciones

Una **condición** es una expresión que combina dos valores con un operador de comparación (tal como `<` o `>`) y se evalúa a un valor Booleano. Una condición es sólo otro nombre para una expresión que se evalúa a `True` o `False`. Las condiciones se usan en sentencias `while` (y en algunas otras situaciones, explicadas más adelante.)

Por ejemplo, la condición `intentosRealizados < 6` pregunta, "¿es el valor almacenado en `intentosRealizados` menor que el número 6?" Si es así, entonces la condición se evalúa a `True` (Verdadero). En caso contrario, la condición se evalúa a `False` (Falso).

En el caso del programa "Adivina el Número", en la línea 4 has almacenado el valor `0` en `intentosRealizados`. Como `0` es menor que `6`, esta condición se evalúa al valor Booleano `True`. La evaluación ocurre así:

```
intentosRealizados < 6
                   ▼
                 0 < 6
                   ▼
                 True
```

## *Experimentando con Booleans, Operadores de Comparación y Condiciones*

Escribe las siguientes expresiones en la consola interactiva para ver sus resultados Booleanos:

```
>>> 0 < 6
True
>>> 6 < 0
False
>>> 50 < 10
False
>>> 10 < 11
True
>>> 10 < 10
False
```

La condición `0 < 6` devuelve el valor Booleano `True` porque el número `0` es menor que el número `6`. Pero como `6` no es menor que `0`, la condición `6 < 0` se evalúa a `False`. `50` no es menor que `10`, luego `50 < 10` es `False`. `10` es menor que `11`, entonces `10 < 11` es `True`.

Observa que `10 < 10` se evalúa a `False` porque el número `10` no es más pequeño que el número `10`. Son exactamente del mismo tamaño. Si Alicia fuera igual de alta que Berto, no dirías que Alicia es más alta que Berto o que Alicia más baja que Berto. Ambas afirmaciones serían falsas.

Ahora prueba introducir estas expresiones en la consola interactiva:

```
>>> 10 == 10
True
>>> 10 == 11
False
>>> 11 == 10
False
```

```
>>> 10 != 10
False
>>> 10 != 11
True
>>> 'Hola' == 'Hola'
True
>>> 'Hola' == 'Adios'
False
>>> 'Hola' == 'HOLA'
False
>>> 'Adios' != 'Hola'
True
```

## La Diferencia Entre = y ==

Intenta no confundir el operador asignación (=) y el operador de comparación "igual a" (==). El signo igual (=) se usa en sentencias de asignación para almacenar un valor en una variable, mientras que el signo igual-igual (==) se usa en expresiones para ver si dos valores son iguales. Es fácil usar uno accidentalmente cuando quieres usar el otro.

Sólo recuerda que el operador de comparación "igual a" (==) está compuesto por dos caracteres, igual que el operador de comparación "diferente a" (`!=`) que también está compuesto por dos caracteres.

Cadenas y valores enteros no pueden ser iguales. Por ejemplo, prueba escribiendo lo siguiente en la consola interactiva:

```
>>> 42 == 'Hola'
False
>>> 42 != '42'
True
```

## Creabdo Bucles con sentencias `while`

La sentencia `while` (mientras) indica el comienzo de un bucle. Los bucles pueden ejecutar el mismo código repetidas veces. Cuando la ejecución llega hasta una sentencia `while`, evalúa la condición junto a la palabra reservada `while`. Si la condición se evalúa a `True`, la ejecución se mueve dentro del bloque `while`. (En tu programa, el bloque while comienza en la línea 13.) Si la condición se evalúa a `False`, la ejecución se mueve hasta debajo del bloque `while`. (En "Adivina el Número", la primera línea luego del bloque `while` es la línea 28.)

Una sentencia `while` siempre incluye dos punos (el signo :) después de la condición.

```
12. while intentosRealizados < 6:
```

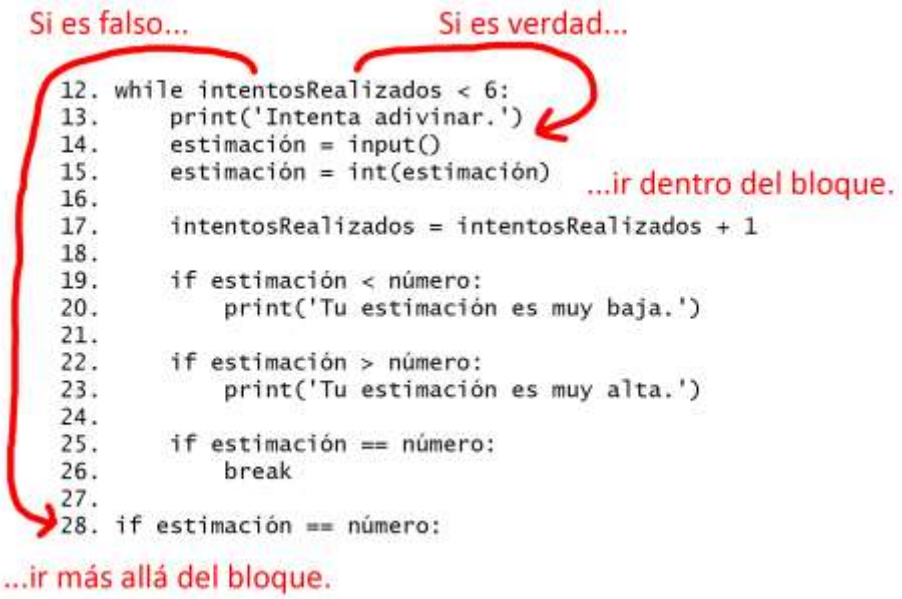



Figura 4-2: La condición del bucle `while`.

La Figura 4-2 muestra como transcurre la ejecución dependiendo de la condición. Si la condición se evalúa a True (lo cual hace la primera vez, porque el valor de `intentosRealizados` es `0`), la ejecución entrará al bloque `while` en la línea 13 y continuará moviéndose hacia abajo. Una vez que el programa llegue al final del bloque `while`, en lugar de ir hacia abajo hasta la siguiente línea, la ejecución vuelve atrás hasta la línea de la sentencia while (línea 12) y reevalúa la condición. Como antes, si la condición es `True` la ejecución vuelve a entrar al bloque `while`. Cada vez que la ejecución recorre el bucle se llama una **iteración**.

Así es como funciona el bucle. Mientras que la condición sea True, el programa sigue ejecutando el código dentro del bloque while en forma repetida hasta la primera vez que la condición sea `False`. Piensa en la sentencia `while` como decir, "mientras esta condición sea verdadera, sigue iterando a través del código en este bloque".

## *El Jugador Adivina*

```
13.     print('Intenta adivinar.') # Hay cuatro espacios delante de print.
14.     estimación = input()
```

Las líneas 13 a 17 piden al jugador que adivine cuál es el númeo secreto y le permiten formular su intento. Este número se almacena en una variable llamada `estimación`.

## Conversión de Cadenas a Enteros con la función `int()`, `float()`, `str()`, `bool()`

```
15.     estimación = int(estimación)
```

En la línea 15, llamas a una función llamada `int()`. La función `int()` toma un argumento y devuelve un valor entero de ese argumento. Prueba escribir lo siguiente en la consola interactiva:

```
>>> int('42')
42
>>> 3 + int('2')
5
```

La llamada a `int('42')` devolverá el valor entero `42`. La llamada `int(42)` hará lo mismo (a pesar de que no tiene mucho sentido obtener la forma de valor entero de un valor que ya es entero). Sin embargo, aunque la función `int()` acepta cadenas, no puedes pasarle cualquier cadena. Pasarle `'cuarenta-y-dos'` a `int()` resultará en un error. La cadena que recibe `int()` debe estar compuesta por números.

```
>>> int('cuarenta-y-dos')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
int('cuarenta-y-dos')
ValueError: invalid literal for int() with base 10: 'cuarenta-y-dos'
```

La línea `3 + int('2')` muestra una expresión que usa el valor de retorno de `int()` como parte de una expresión. Se evalúa al valor entero `5`:

```
3 + int('2')
  ▼
3 + 2
  ▼
  5
```

Recuerda, la función input() devuelve **una cadena** de texto que el jugador ha escrito. Si el jugador escribe `5`, la función `input()` devolverá el valor de cadena `'5'`, no el valor entero `5`. Python no puede usar los operadores de comparación < y > para comparar una cadena y un valor entero:

```
>>> 4 < '5'
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    4 < '5'
TypeError: unorderable types: int() < str()
```

```
14.     estimación = input()
15.     estimación = int(estimación)
```

En la línea 14 la variable `estimación` contenía originalmente el valor de cadena ingresado por el jugador. La línea 15 sobrescribe el valor de cadena en `estimación` con el valor entero devuelto por `int()`. Esto permite al código más adelante en el programa comparar si `estimación` es mayor, menor o igual al `número` secreto en la variable número.

Una última cosa: La llamada `int(estimación)` no cambia el valor de la variable `estimación`. El código `int(estimación)` es una expresión que se evalúa a la forma de valor entero de la cadena guardada en la variable `estimación`. Lo que cambia `estimación` es la sentencia de asignación: `estimación = int(estimación)`

El `float()`, `str()`, y `bool()` funciona de manera similar se volverá float, str, y las versiones de Boole de los argumentos que se pasan a ellos:

```
>>> float('42')
42.0
>>> float(42)
42.0
>>> str(42)
'42'
>>> str(42.0)
'42.0'
>>> str(False)
'False'
>>> bool('')
False
>>> bool('any nonempty string')
True
```

## *Incrementando las Variables*

```
17.     intentosRealizados = intentosRealizados + 1
```

Una vez que el jugador ha realizado un intento, el número de intentos debería incrementarse en uno.

En la primera iteración del bucle, `intentosRealizados` tiene el valor `0`. Python tomará este valor y le sumará 1. `0 + 1` se evalúa a `1`, el cual se almacena como nuevo valor de `intentosRealizados`. Piensa en la línea 17 como diciendo, "la variable `intentosRealizados` debería ser uno más que lo que es ahora".

Sumarle uno al valor entero o float de una variable es lo que se llama **incrementar** la variable. Restarle uno al valor entero o float de una variable es **decrementar** la variable.

## Sentencias `if`

```
19.     if estimación < número:
20.         print('Tu estimación es muy baja.') # Hay ocho espacios delante de
print.
```

La línea 19 es una sentencia `if`. La ejecución correrá el código en el siguiente bloque si la condición de la sentencia `if` se evalúa a `True`. Si la condición es `False`, entonces el código en el bloque `if` se omite. Mediante el uso de sentencias `if`, puedes hacer que el programa sólo ejecute ciertas partes del código cuando tú quieras.

La sentencia `if` funciona casi igual que una sentencia while. Pero a diferencia del bloque while, la ejecución no vuelve atrás hasta la sentencia `if` cuando termina de ejecutarse el bloque if. Simplemente continúa en la línea siguiente. En otras palabras, las sentencias `if` no generan un bucle. Mira la Figura 4-3 para ver una comparación de las dos sentencias.

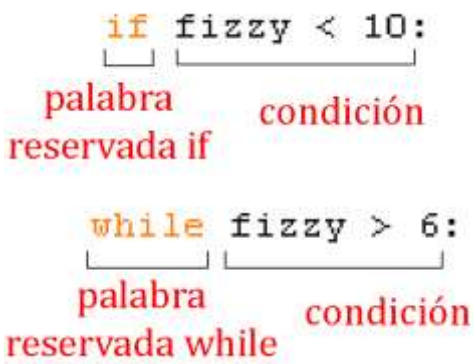



Figura 4-3: Sentencias `if` y `while`.

```
22.     if estimación > número:
23.         print('Tu estimación es muy alta.')
```

La línea 22 comprueba si la estimación del jugador es mayor que el entero aleatorio. Si esta condición es `True`, entonces la llamada a la función `print()` indica al jugador que su estimación es demasiado alta.

## Abandonando los Bucles Anticipadamente con la sentencia `break`

```
25.     if estimación == número:
26.         break
```

La sentencia if en la línea 25 comprueba si la estimación es igual al entero aleatorio. Si lo es, el programa ejecuta la sentencia break de la línea 26.

Una sentencia **`break`** indica a la ejecución que salga inmediatamente del bucle `while` y se mueva a la primera línea a continuación del mismo. (Las sentencias break no se molestan en volver a revisar la condición del bucle `while`, sólo salen del bucle instantaneamente.)

La sentencia `break` es simplemente la palabra reservada `break` en sí misma, sin condición o dos puntos.

Si el jugador adivinó el número no es igual al número entero aleatorio, la ejecución alcanza la parte inferior del bloque `while`. Esto significa se repetirá la ejecución de nuevo a la parte superior y vuelva a comprobar el estado de la línea 12 (`intentosRealizados < 6`). Recuerdo que después de los `intentosRealizados = intentosRealizados + 1` línea de código se ejecuta, el nuevo valor de `intentosRealizados` es `1`. Porque `1 <6` es cierto que la ejecución entra en el bucle de nuevo.

Si el jugador continúa realizando intentos demasiado altos o bajos, el valor de `intentosRealizados` cambiará a `2`, luego `3`, luego `4`, luego `5`, luego `6`. Cuando `intentosRealizados` tiene almacenado el número `6`, la condición de la sentencia while es `False`, dado que `6` no es menor que `6`. Como la condición de la sentencia `while` es `False`, la ejecución se mueve a la primera línea después del bloque `while`, línea 28.

### *Comprobar si el Jugador ha Ganado*

```
28. if estimación == número:
```

La línea 28 no tiene indentación, lo que significa que el bloque `while` ha terminado y esta es la primera línea luego del mismo. La ejecución ha abandonado el bloque `while`, sea porque la condición de la sentencia while era `False` (cuando el jugador se quedó sin intentos) o porque se ejecutó la sentencia break (cuando el jugador adivina el número correctamente).

La línea 28 comprueba a ver si el jugador ha adivinado correctamente. Si es así, la ejecución entra al bloque `if` de la línea 29.

```
29.     intentosRealizados = str(intentosRealizados)
30.     print('¡Buen trabajo, ' + miNombre + '! ¡Has adivinado mi número en ' +
intentosRealizados + ' intentos!')
```

Las líneas 29 y 30 sólo se ejecutan si la condición en la sentencia if de la línea 28 es `True` (es decir, si el jugador ha adivinado correctamente el número de la computadora).

La línea 29 llama a la nueva función `str()`, que devuelve la forma cadena de `intentosRealizados`. Este código obtiene la forma cadena del entero en `intentosRealizados` ya que sólo cadenas pueden ser concatenadas con otras cadenas.

## *Comprobar si el Jugador ha Perdido*

```
32. if estimación != número:
```

La línea 32 usa el operador comparación `!=` para comprobar si el último intento del jugador no es igual al número secreto. Si esta condición se evalúa a `True`, la ejecución se mueve dentro del bloque if de la línea 33.

Las líneas 33 y 34 están dentro del bloque `if`, y sólo se ejecutan si la condición de la línea 32 es `True`.

```
33.     número = str(número)
34.     print('Pues no. El número que estaba pensando era ' + número)
```

En este bloque, el programa indica al jugador cuál era el número secreto que no ha podido adivinar correctamente. Esto requiere concatenar cadenas, pero número almacena un valor entero. La línea 33 reemplazará `número` con una forma cadena, de modo que pueda ser concatenada con la cadena `'Pues no. El número que estaba pensando era '` de la línea 34.

En este punto, la ejecución ha alcanzado el final del código, y el programa termina. ¡Felicitaciones! ¡Acabas de programar tu primer juego de verdad!

Puedes cambiar la dificultad del juego modificando el número de intentos que el jugador recibe. Para dar al jugador sólo cuatro intentos, cambia esta línea::

```
12. while intentosRealizados < 6:
```

...por esta otra:

```
12. while intentosRealizados < 4:
```

El código más adelante en el bloque while incrementa la variable `intentosRealizados` en `1` en cada iteración. Al imponer la condición `intentosRealizados < 4`, te aseguras de que el código dentro del bucle sólo se ejecuta cuatro veces en lugar de seis. Esto hace al juego mucho más difícil. Para hacer el juego más fácil, cambia la condición a `intentosRealizados < 8` o `intentosRealizados < 10`. Esto permitirá que el bucle se ejecute algunas veces *más* y acepte *más* intentos del jugador.

## Sentencias de Control de Flujo

En capítulos anteriores, la ejecución del programa comenzaba por la instrucción de más arriba e iba directo hacia abajo, ejecutando cada instrucción en orden. Pero con las sentencias `while`, `if`, `else`, y `break`, puedes hacer que la ejecución repita u omita instrucciones basándose en condiciones. Este tipo de sentencia se llama **sentencia de control de flujo**, ya que modifican el "flujo" de la ejecución a medida que esta se desplaza por tu programa.

### *Resumen*

Si alguien te preguntase **"¿Qué es exactamente programar de todos modos?"**, ¿qué podrías decirle? Programar es simplemente la acción de escribir código para programas, es decir, crear programas que puedan ser ejecutados por una computadora.

**"Pero ¿qué es exactamente un programa?"** Cuando ves a alguien usando un programa de computadora (por ejemplo, jugando tu juego "Adivina el Número"), todo lo que ves es texto apareciendo en la pantalla. El programa decide exactamente qué texto mostrar en la pantalla (**las salidas** del programa), basado en instrucciones y en el texto que el jugador ha escrito mediante el teclado (**las entradas** del programa). Un programa es sólo una colección de instrucciones que actúan sobre las entradas provistas por el usuario.

**"¿Qué tipo de instrucciones?"** Hay sólo unos pocos tipos diferentes de instrucciones, de verdad.

1. Expresiones. Las expresiones son valores conectados por operadores. Todas las expresiones son evaluadas a un único valor, así como `2 + 2` se evalúa a `4` o `'Hola' + ' ' + 'Mundo'` se evalúa a `'Hola Mundo'`. Cuando las expresiones están al lado de las palabras reservadas `if` y `while`, pueden recibir también el nombre de condiciones.
2. **Sentencias de asignación**. Las sentencias de asignación almacenan valores en variables para que puedas recordar los valores más adelante en el programa.
3. **Sentencias de control de flujo `if`, `while`, y `break`.** Las sentencias de control de flujo pueden hacer que el flujo omita instrucciones, genere un bucle sobre un bloque de

instrucciones o salga del bucle en el que se encuentra. Las llamadas a funciones también cambian el flujo de ejecución moviéndose al comienzo de una función.

4. **Las funciones** `print()` **e** `input()`**.** Estas funciones muestran texto en la pantalla y reciben texto del teclado. Esto se llama **E/S** (o en inglés I/O), porque tiene que ver con las Entradas y Salidas del programa.

Y eso es todo, sólo estas cuatro cosas. Por supuesto, hay muchos detalles acerca de estos cuatro tipos de instrucciones. En este libro aprenderás acerca de nuevos tipos de datos y operadores, nuevas sentencias de controlo de flujo, y muchas otras funciones que vienen con Python. También hay diferentes tipos de E/S tales como entradas provistas por el ratón o salidas de sonido y gráficos en lugar de sólo texto.

En cuanto a la persona que usa tus programas, sólo se preocupa acerca del último tipo, E/S. El usuario escribe con el teclado y luego ve cosas en la pantalla u oye sonidos de los altavoces. Pero para que la computadora pueda saber qué imágenes mostrar y qué sonidos reproducir, necesita un programa, y los programas son sólo un manojo de instrucciones que tú, el programador, has escrito.

# Capítulo 5
# CHISTES

Temas Tratados En Este Capítulo:
- Caracteres de escape
- Utilizando comillas simples y comillas dobles para las cadenas.
- Utilizando el argumento palabra clave final (end) de print() para evitar nuevas lineas

## Aprovechar `print()` al Máximo

La mayoría de los juegos en este libro tendrán un texto simple de entrada y salida. La entrada es escrita por el usuario desde el teclado e introducida a la computadora. La salida es el texto mostrado en la pantalla. En Python, la función `print()` se puede usar para mostrar salidas de texto en la pantalla. Pero hay mas para aprender sobre cómo funcionan las cadenas y el `print()` en Python.

El programa de este capítulo le cuenta chistes al usuario.

## Ejecución de Muestra de Chistes

```
¿Qué sale de la cruza entre un mono y un pato?
¡Un monopatín!
¿Porqué vuelan los pájaros pa'l sur?
¡Porque caminando tardarían muchísimo!
¿En qué se parecen una familia, un bombero y un barco?
No sé... ¿en qué se parecen?
En que el bombero y el barco tienen casco.
¿Y la familia? -Bien, gracias.
```

## Source Code of Jokes

Escriba el siguiente código fuente en el editor de archivos y guárdelo como *chistes.py*.

> **¡NOTA IMPORTANTE!** Los programas de este libro sólo podrán ejecutarse sobre Python 3, no Python 2. Al iniciar la ventana IDLE, dirá algo como “Python 3.4.2” en la parte superior. Si tienes Python 2 instalado, es posible instalar también Python 3 a la vez. Para descargar Python 3, dirígete a https://python.org/download/.

Si obtiene errores después de escribir este código, compárelo con el código del libro con la herramienta diff en línea en http://invpy.com/es/diff/chistes.

jokes.py

```
 1. print('¿Qué sale de la cruza entre un mono y un pato?')
 2. input()
 3. print('¡Un monopatín!')
 4. print()
 5. print('¿Porqué vuelan los pájaros pa\'l sur?')
 6. input()
 7. print('¡Porque caminando tardarían muchísimo!')
 8. print()
 9. print('¿En qué se parecen una familia, un bombero y un barco?')
10. input()
11. print("No sé... ¿en qué se parecen?")
12. input()
13. print('En que el bombero y el barco tienen casco.')
14. input()
15. print('¿Y la familia?', end='')
16. print(' -Bien, gracias.')
```

## *Cómo Funciona el Código*

```
1. print('¿Qué sale de la cruza entre un mono y un pato?')
2. input()
3. print('¡Un monopatín!')
4. print()
```

Las líneas de la 1 a la 4 tienen tres llamadas a la función `print()`. No quieres que el jugador lea de inmediato el remate del chiste, así que hay una llamada a la función `print()` después del primer `print()`. El jugador puede leer la primera línea, presionar INTRO, y entonces leer el remate del chiste.

El usuario todavía puede escribir una cadena y pulsar INTRO, pero esta cadena devuelta no está siendo almacenada en ninguna variable. El programa tan solo lo olvidará y se moverá a la siguiente línea de código.

La última llamada a la función `print()` no tiene argumento de cadena. Esto le indica al programa que solamente escriba una línea en blanco. Las líneas en blanco pueden ser útiles para evitar que el texto quede unido.

## Caracteres de Escape

```
5. print('¿Porqué vuelan los pájaros pa\'l sur?')
6. input()
7. print('¡Porque caminando tardarían muchísimo!')
8. print()
```

En el primer `print()` de arriba, ha una barra invertida justo antes de la comillas simple (esto es, el apóstrofo). Note que \ es una barra inversa, y / es una barra inclinada. Esta barra inversa indica que la letra que está a su derecha es una caracter de escape. Un **caracter de escape** te permite imprimir caracteres que son difíciles de introducir en el código fuente. En esta llamada a `print()` el caracter de escape es una comilla simple.

El caracter de escape comilla simple está allí porque de otra manera Python pensaría que la comilla indica el final de la cadena. Pero esta comilla necesita *formar parte de* la cadena. La comilla simple de escape le indica a Python que la comilla simple es literalmente una parte de la cadena en lugar de indicar el final del valor de la cadena.

### *Algunos Otros Caracteres de Escape*

¿Qué pasa si realmente quisieras escribir una barra invertida?. Esta línea de código no funcionaría:

```
>>> print('Él se fue volando en un helicóptero verde\turquesa.')
Él se fue volando en un helicóptero verde    urquesa.
```

Esto es porque la "t" en "turquesa" fue vista como un caracter de escape debido a que estaba después de una barra inversa. El caracter de escape t simula la pulsación de la tecla TAB de tu teclado. Hay caracteres de escape para que las cadenas puedan tener caracteres que no se pueden escribir.

En lugar de eso, prueba con esta línea:

```
>>> print('Él se fue volando en un helicóptero verde\\turquesa.')
Él se fue volando en un helicóptero verde\turquesa.
```

La tabla 5-1 es una lista de caracteres de escape en Python.

Tabla 5-1: Caracteres de Escape

| Caracter de Escape | Lo Que Imprime |
|---|---|
| \\ | Barra inversa (\) |
| \' | Comilla simple (') |
| \" | Comilla doble (") |
| \n | Salto de línea |
| \t | Tabulador |

## Comillas Simples y Dobles

La cadenas en Python no tienen que estar siempre entre comillas simples. También puedes ponerlas entre comilas dobles. Estados dos líneas imprimen lo mismo:

```
>>> print('Hola mundo')
Hola mundo
>>> print("Hola mundo")
Hola mundo
```

Pero no puedes mexzclar las comillas. Esta línea devolverá un error si intentas utilizarla:

```
>>> print('Hola mundo")
SyntaxError: EOL while scanning single-quoted string
```

Me gusta utilizar las comillas simples, así no tengo que pulsar la tecla shift (mayúsculas) para escribirlas. Es más fácil de escribir, y a Python le da igual de cualquier manera.

Del mismo modo en que necesitas el caracter de escape \' para obtener una comilla simple en una cadena rodeada de comillas simples, se necesita un caracter de escape \" para imprimir una comilla doble en una cadena rodeada de comillas dobles. Por ejemplo, mira estas dos líneas:

```
>>> print('Le pedí prestado el carro a Pedro pa\'ir al pueblo. El dijo,
"Seguro."')
Le pedí prestado el carro a Pedro pa'ir al pueblo. El dijo, "Seguro."

>>> print("Él dijo, \"No puedo creer que lo dejaste llevarse el carro pa'l
pueblo\"")
Él dijo, "No puedo creer que lo dejaste llevarse el carro pa'l pueblo"
```

En las cadenas de comillas simples no necesitas escapar las comillas dobles, y en las cadenas de comillas dobles no necesitas escapar las comillas simples. El intérprete de Python tiene inteligencia suficiente para saber que si una cadena comienza con un tipo de comillas, el otro tipo de comillas no significa que la cadena está terminando.

## El Argumento de Palabra end

```
 9. print('¿En qué se parecen una familia, un bombero y un barco?')
10. input()
11. print("No sé... ¿en qué se parecen?")
12. input()
13. print('En que el bombero y el barco tienen casco.')
14. input()
15. print('¿Y la familia?', end='')
16. print(' -Bien, gracias.')
```

¿Te diste cuenta del segundo parámetro en el print de la línea 15?. Normalmente, `print()` añade un salto de línea al final de la cadena que imprime. Por esta razón, una función `print()` en blanco tan solo imprimirá una nueva línea. Pero la función `print()` tiene la opción de un segundo parámetro (que tiene nombre "end" (fin)).

La cadena en blanco dada se llama **argumento de palabra clave**. El parámetro final tiene un nombre específico, y para pasar un argumento a ese parámetro en particular necesitamos utilizar la sintáxis `end=`.

Pasando una cadena en blanco usando end, la función `print()` no añadirá un salto de linea al final de la cadena, en lugar de esto añadirá una cadena en blanco. Por esta razón `' -Bien, gracias.'` aparece junto a la línea anterior, en lugar una línea nueva aparte. No hubo salto de línea después de la cadena `'¿Y la familia?'`.

### *Resumen*

Este capítulo explora las diferentes formas en las que se puede utilizar la función `print()`. Los caracteres de escape se utilizan para los caracteres que son difíciles o imposibles de escribir en código usando el teclado. Los caracteres de escape se escriben en las cadenas comienzando con una barra inversa \ seguida de una sola letra para el carácter de escape. Por ejemplo, `\n` sería un salto de línea. Para incluir una barra invertida en una cadena, deberás utilizar el carácter de escape \\.

La función `print()` añade automáticamente un carácter de salto de línea al final de la cadena que se pasa para imprimr en pantalla. La mayor parte del tiempo, es un atajo útil. Pero a veces no quieres un carácter de salto de línea al final. Para cambiar esto, puedes pasar el argumento de palabra clave end con una cadena en blanco. Por ejemplo, para imprimir "spam" en la pantalla sin un carácter de salto de línea, podrías hacer el llamado `print('spam', end='')`.

Al añadir este nivel de control sobre el texto que mostraremos en la pantalla, puedes tener formas más flexibles para hacerlo.

# Capítulo 6

# REINO DE DRAGONES

Temas Tratados En Este Capítulo:
- La función `time.sleep()`
- Creando nuestras propias funciones con la palabra reservada `def`
- La palabra reservada `return`
- Los operadores Booleanos `and`, `or` y `not`
- Tablas de verdad
- Entorno de variables (Global y Local)
- Parámetros y Argumentos
- Diagramas de Flujo

## Las Funciones

Ya hemos usado dos funciones en nuestros programas anteriores: `input()` y `print()`. En los programas anteriores, hemos llamado a estas funciones para ejecutar el código dentro de ellas. En este capítulo, escribiremos nuestras propias funciones para que sean llamadas por programas. Una función es como un mini-programa dentro de nuestro programa.

El juego que crearemos para presentar las funciones se llama "Reino de Dragones", y permite al jugador elegir entre dos cuevas, en una de las cuales encontrará un tesoro y en la otra su perdición.

## Cómo Jugar a Reino de Dragones

En este juego, el jugador está en una tierra llena de dragones. Todos los dragones viven en cuevas junto a sus grandes montones de tesoros encontrados. Algunos dragones son amigables, y compartirán sus tesoros contigo. Otros son codiciosos y hambrientos, y se comerán a cualquiera que entre a su cueva. El jugador se encuentra frente a dos cuevas, una con un dragón amigable y la otra con un dragón hambriento. El jugador tiene que elegir entre las dos.

Abre una nueva ventana del editor de archivos haciendo clic en el menú **File** (Archivo) ► **New Window** (Nueva Ventana). En la ventana vacía que aparece escribe el código fuente y guárdalo como *dragón.py*. Luego ejecuta el programa pulsando F5.

## Prueba de Ejecución de Reino de Dragones

```
Estás en una tierra llena de dragones. Frente a tí
hay dos cuevas. En una de ellas, el dragón es generoso y amigable
y compartirá su tesoro contigo. El otro dragón
es codicioso y está hambriento, y te devorará inmediatamente.
¿A qué cueva quieres entrar? (1 ó 2)
1
Te aproximas a la cueva...
Es oscura y espeluznante...
¡Un gran dragon aparece súbitamente frente a tí! Abre sus fauces y...
¡Te engulle de un bocado!
¿Quieres jugar de nuevo? (sí or no)
no
```

## El Código Fuente de Reino de Dragones

> **¡NOTA IMPORTANTE!** Los programas de este libro sólo podrán ejecutarse sobre Python 3, no Python 2. Al iniciar la ventana IDLE, dirá algo como "Python 3.4.2" en la parte superior. Si tienes Python 2 instalado, es posible instalar también Python 3 a la vez. Para descargar Python 3, dirígete a https://python.org/download/.

If you get errors after typing this code in, compare the code you typed to the book's code with the online diff tool at http://invpy.com/diff/dragón.

dragón.py

```
 1. import random
 2. import time
 3.
 4. def mostrarIntroducción():
 5.     print('Estás en una tierra llena de dragones. Frente a tí')
 6.     print('hay dos cuevas. En una de ellas, el dragón es generoso y')
 7.     print('amigable y compartirá su tesoro contigo. El otro dragón')
 8.     print('es codicioso y está hambriento, y te devorará inmediatamente.')
 9.     print()
10.
11. def elegirCueva():
12.     cueva = ''
13.     while cueva != '1' and cueva != '2':
14.         print('¿A qué cueva quieres entrar? (1 ó 2)')
15.         cueva = input()
16.
17.     return cueva
```

```
18.
19. def explorarCueva(cuevaElegida):
20.     print('Te aproximas a la cueva...')
21.     time.sleep(2)
22.     print('Es oscura y espeluznante...')
23.     time.sleep(2)
24.     print('¡Un gran dragon aparece súbitamente frente a tí! Abre sus fauces
y...')
25.     print()
26.     time.sleep(2)
27.
28.     cuevaAmigable = random.randint(1, 2)
29.
30.     if cuevaElegida == str(cuevaAmigable):
31.         print('¡Te regala su tesoro!')
32.     else:
33.         print('¡Te engulle de un bocado!')
34.
35. jugarDeNuevo = 'sí'
36. while jugarDeNuevo == 'sí' or jugarDeNuevo == 's':
37.
38.     mostrarIntroducción()
39.
40.     númeroDeCueva = elegirCueva()
41.
42.     explorarCueva(númeroDeCueva)
43.
44.     print('¿Quieres jugar de nuevo? (sí o no)')
45.     jugarDeNuevo = input()
```

### *Cómo Funciona el Código*

Veamos el código fuente en más detalle.

```
1. import random
2. import time
```

El programa importa dos módulos. El módulo random proveerá la función `random.randint()` como lo hizo en el juego "Adivina el Número". También precisarás funciones relacionadas con tiempo, que están incluidas en el módulo `time`, de modo que también importaremos este módulo.

## Sentencias `def`

```
4. def mostrarIntroducción():
```

```
5.     print('Estás en una tierra llena de dragones. Frente a tí')
6.     print('hay dos cuevas. En una de ellas, el dragón es generoso y')
7.     print('amigable y compartirá su tesoro contigo. El otro dragón')
8.     print('es codicioso y está hambriento, y te devorará inmediatamente.')
9.     print()
```

La línea 4 es una sentencia `def`. La **sentencia def** crea, es decir, una nueva función que puede ser llamada más adelante en el programa. Luego de *haber definido* esta función, puedes llamarla de la misma forma en que llamas a otras funciones. Cuando *llamas* a esta función, el código dentro del bloque `def` se ejecuta.

La Figura 6-1 muestra las partes de una sentencia `def`. Comienza con la palabra reservada `def` seguida por un nombre de función con paréntesis y luego dos puntos. El bloque a continuación de la sentencia def se llama el bloque `def`.


Figura 6-1: Las partes de una sentencia `def`.

Recuerda, la sentencia def no ejecuta el código. Sólo define qué código se ejecutará cuando llames a la función. Cuando la ejecución llega a una sentencia `def`, omite lo que sigue hasta la primera línea a continuación del bloque `def`.

Pero cuando la función `mostrarIntroducción()` es llamada (como en la línea 38), la ejecución entra a la función `mostrarIntroducción()` y se posiciona en la primera línea del bloque `def`.

```
38.    mostrarIntroducción()
```

Entonces todas las llamadas a `print()` se ejecutan, y se muestra la introducción "Estás en una tierra llena de dragones...".

## *Dónde Colocar las Definiciones de Funciones*

La sentencia def y el bloque `def` de una función deben aparecer *antes* de llamar a la función. Esto es igual que cuando tienes que asignarle un valor a una variable antes de usar la variable. Si colocas la llamada a la función antes que la definición de la función, obtendrás un error. Por ejemplo, mira este código:

```
decirAdios()

def decirAdios():
    print('¡Adios!')
```

Si tratas de ejecutarlo, Python te dará un mensaje de error como este:

```
Traceback (most recent call last):
  File "C:\Python34\spam.py", line 1, in <module>
decirAdios()
NameError: name 'decirAdios' is not defined
```

Para arreglar esto, coloca la definición de la función antes de llamar a la función:

```
def sayGoodbye():
    print('Goodbye!')

sayGoodbye()
```

### *Definiendo la Función `elegirCueva()`*

```
11. def elegirCueva():
```

La línea 11 define otra función llamada `elegirCueva()`. El código de esta función pregunta al jugador a qué cueva quiere entrar, 1 ó 2.

```
12.     cueva = ''
13.     while cueva != '1' and cueva != '2':
```

Esta función necesita asegurar que el jugador haya respondido 1 ó 2, y no otra cosa. Un bucle aquí seguirá preguntando al jugador hasta que escriba alguna de estas dos respuestas válidas. Esto se llama **validación de entrada**.

La línea 12 crea una nueva variable llamada `cueva` y guarda en ella una cadena vacía. Luego un bucle `while` comienza en la línea 13. La condición contiene un nuevo operador que no has visto antes llamado `and` (y). Igual que los signos `-` o `*` son operadores matemáticos y los signos `==` o `!=` son operadores de comparación, el operador `and` es un operador Booleano.

## Operadores Booleanos

La lógica Booleana se ocupa de enunciados que son verdaderas (`True`) o falsos (`False`). Los operadores Booleanos comparan dos valores Booleanos y se evalúan a un único valor Booleano.

Piensa en este enunciado, “Los gatos tienen bigotes y los perros tienen colas.” “Los gatos tienen bigotes” es verdadero y “los perros tienen colas” también es verdadero, luego el enunciado completo “Los gatos tienen bigotes **y** los perros tienen colas” es verdadero.

Pero el enunciado “Los gatos tienen bigotes y los perros tienen alas” sería falso. Incluso si “los gatos tienen bigotes” es verdadero, los perros no tienen alas, luego “los perros tienen alas” es falso. En lógica Booleana, los enunciados sólo pueden ser completamente verdaderos o completamente falsos. Debido a la conjunción “y”, el enunciado completo es verdadero sólo **si ambas partes** son verdaderas. Si una o ambas partes son falsas, entonces el enunciado completo es falso.

## *Los operadores and y or*

El operador `and` en Python es igual que la conjunción “y”. Si los valores Booleanos a ambos lados de la palabra reservada and son `True`, entonces la expresión se evalúa a `True`. Si alguno o ambos valores Booleanos es `False`, la expresión se evalúa a `False`.

Prueba escribir las siguientes expresiones con el operador and en la consola interactiva:

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
>>> spam = 'Hola'
>>> 10 < 20 and spam == 'Hola'
True
```

El operador `or` es similar al operador and, excepto que se evaluará a `True` si *cualquiera de los dos* valores Booleanos es `True`. La única vez en que el operador or se evalúa a `False` es si *los dos* valores Booleanos son `False`.

Prueba escribir lo siguiente en la consola interactiva:

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

```
>>> 10 > 20 or 20 > 10
True
```

## El Operador `not`

El operador `not` sólo actúa sobre un valor, en lugar de combinar dos valores. El operador `not` (no) se evalúa al valor Booleano opuesto. La expresión `not True` se evaluará a `False` y `not False` se evaluará a `True`.

Prueba escribir lo siguiente en la consola interactiva:

```
>>> not True
False
>>> not False
True
>>> not ('negro' == 'blanco')
True
```

## Tablas de Verdad

Si alguna vez te olvidas cóno funcionan los operadores Booleanos, puedes mirar estas **tablas de verdad**:

Tabla 6-1: La tabla de verdad del operador and.

| A | and | B | is | Enunciado completo |
|---|---|---|---|---|
| True | and | True | es | True |
| True | and | False | es | False |
| False | and | True | es | False |
| False | and | False | es | False |

Tabla 6-2: La tabla de verdad del operador or.

| A | or | B | es | Enunciado completo |
|---|---|---|---|---|
| True | or | True | es | True |
| True | or | False | es | True |
| False | or | True | es | True |
| False | or | False | es | False |

Table 6-3: La tabla de verdad del operador not.

| not A | es | Enunciado completo |
|---|---|---|
| not True | es | False |
| not False | es | True |

## Evaluando Operadores Booleanos

Miremos otra vez la línea 13:

```
13.     while cueva != '1' and cueva != '2':
```

La condición tiene dos partes conectadas por el operador Booleano `and`. La condición es `True` sólo si ambas partes son `True`.

La primera vez que se comprueba la condición de la sentencia `while`, `cueva` está definida como la cadena vacía, `''`. La cadena vacía no es igual a la cadena `'1'`, luego el lado izquierdo se evalúa a `True`. La cadena vacía tampoco es igual a la cadena `'2'`, por lo que el lado derecho se evalúa a `True`.

Entonces la condición se transforma en `True and True`. Como ambos valores Booleanos son `True`, la condición finalmente se evalúa a `True`. Luego la ejecución del programa entra al bloque `while`.

Así es como se ve la evaluación de la condición (si el valor de `cueva` es la cadena vacía):

```
while cueva != '1' and cueva != '2':
                    ▼
while  ''  != '1'  and cueva != '2':
                    ▼
while     True     and cueva != '2':
                    ▼
while     True     and  ''  != '2':
                    ▼
while     True     and     True:
                    ▼
while              True:
```

## Obteniendo la Entrada de Datos del Jugador

```
13.     while cueva != '1' and cueva != '2':
14.         print('¿A qué cueva quieres entrar? (1 ó 2)')
15.         cueva = input()
```

La línea 14 pregunta al jugador qué cueva quiere elegir. La línea 15 permite al jugador escribir la respuesta y pulsar INTRO. Esta respuesta es almacenada en cueva. Después de ejecutar este

código, la ejecución vuelve a la parte superior de la sentencia while y vuelve a comprobar la condición.

Si el jugador ha ingresado `1` ó `2`, entonces cueva será `'1'` or `'2'` (ya que `input()` siempre devuelve cadenas). Esto hace que la condición sea `False`, y la ejecución del programa continuará debajo del bucle `while`. Por ejemplo, si el usuario escribiese `'1'` la evaluación se vería así:

```
while cueva != '1' and cueva != '2':
                   ▼
while   '1' != '1' and cueva != '2':
                   ▼
while      False   and cueva != '2':
                   ▼
while      False   and  '1' != '2':
                   ▼
while      False   and     True:
                   ▼
while             False:
```

Pero si el jugador hubiese escrito 3 o 4 o HOLA, esa respuesta habría sido inválida. La condición seguiría siendo True y entrando al bloque `while` para preguntar de nuevo al jugador. El programa simplemente continúa preguntando hasta que el jugador responda 1 or 2. Esto garantiza que cuando la ejecución continúe avanzando la variable cueva contendrá una respuesta válida.

## Retorno de Valores

```
17.     return cueva
```

Esta es una **sentencia return**, la cual sólo aparece dentro de bloques `def`. ¿Recuerdas como la función `input()` devuelve un valor de cadena que el jugador ha ingresado? La función `elegirCueva()` también devuelve un valor. La línea 17 devuelve la cadena almacenada en cueva, sea `'1'` o `'2'`.

Una vez ejecutada la sentencia `return`, la ejecución del programa sale inmediatamente del bloque `def`. (Esto es como cuando la sentencia `break` hace que la ejecución salga de un bloque `while`.) La ejecución del programa vuelve a la línea que contiene la llamada a la función. La llamada a la función será entonces evaluada al valor de retorno.

Ve ahora hacia abajo y observa la línea 40 por un momento:

```
40.     númeroDeCueva = elegirCueva()
```

Cuando `elegirCueva()` es llamada más adelante por el programa en la línea 40, el valor de retorno es almacenado en la variable `númeroDeCueva`. El bucle while garantiza que `elegirCueva()` devolverá sólo `'1'` o `'2'` como valor de retorno.

Entonces cuando la línea 17 devuelve una cadena, la llamada a la función en la línea 40 es evaluada a esa cadena, la cual se almacena en `númeroDeCueva`.

## Entorno Global y Entorno Local

Las variables de tu programa son olvidadas en cuanto el programa termina. Lo mismo ocure con estas variables creadas mientras la ejecución está dentro de la llamada a una función. Las variables se crean cuando la función es llamada y se olvidan cuando la función devuelve un valor. Recuerda, las funciones son como mini-programas dentro de tu programa.

Cuando la ejecución está dentro de una función, no puedes modificar las variables fuera de la función, incluidas variables de otras funciones. Esto es porque esas variables existen en un "entorno" diferente. todas las variables existen en el entorno global o en el entorno local de la llamada a una función.

El entorno exterior a todas las funciones se llama **entorno global**. El entorno dentro de una función (por la duración de una llamada específica a la función) se llama **entorno local**.

El programa entero tiene un solo entorno global. Las variables definidas en el entorno global puede ser leídas fuera y dentro de las funciones, pero sólo pueden ser modificadas fuera de todas las funciones. Las variables creadas en la llamada a una función sólo pueden ser leídas o modificadas durante esa llamada a la función.

Puedes leer el valor de las variables globales desde el entorno local, pero intentar modificar una variable global desde el entorno local no funcionará. Lo que Python hace en ese caso es crear una variable local **con el mismo nombre** que la variable global. Sería posible, por ejemplo, tener una variable local llamada spam al mismo tienpo que existe una variable global llamada spam. Python las considerará dos variables distintas.

Mira el siguiente ejemplo para ver qué pasa cuando intentas modificar una variable global desde dentro de un entorno local. Los comentarios explican qué es lo que está ocurriendo:

```
def bacon():
    # Creamos una variable local llamada "spam"
    # en lugar de cambiar el valor de la
    # variable global "spam":
    spam = 99
    # El nombre "spam" se refiere ahora sólo a la
    # variable local por el resto de esta
    # función:
```

```
    print(spam)    # 99

spam = 42 # Una variable global llamada "spam":
print(spam) # 42
bacon() # Llama a la función bacon():
# La variable global no fue cambiada en bacon():
print(spam)    # 42
```

Al ser ejecutado, este código mostrará las siguientes salidas:

```
42
99
42
```

Dónde se crea una variables determina en qué entorno se encuentra. Cuando el programa Reino de Dragones ejecuta por primera vez la línea:

```
12.     cueva = ''
```

...la variable cueva se crea dentro de la función `elegirCueva()`. Esto significa que es creada en el entorno local de la función `elegirCueva()`. Será olvidada cuando `elegirCueva()` finalice, y será recreada si `elegirCueva()` es llamada por segunda vez. El valor de una variable local no es recordado entre una llamada a una función local y otra.

## Parámetros

```
19. def explorarCueva(cuevaElegida):
```

La siguiente función que el programa define se llama `explorarCueva()`. Nota el texto `cuevaElegida` entre paréntesis. Esto es un **parámetro**: una variable local a la que se asigna el argumento pasado cuando esta función es llamada.

Recuerda cómo para algunas llamadas a funciones como `str()` o `randint()`, pasarías un argumento entre paréntesis:

```
>>> str(5)
'5'
>>> random.randint(1, 20)
14
```

También pasarás un argumento al llamar a `explorarCueva()`. Este argumento es almacenado en una nueva variable llamada `cuevaElegida`. Estas variables también se denominan parámetros.

Por ejemplo, aquí hay un pequeño programa que demuestra cómo se define una función con un parámetro:

```
def decirHola (nombre):
    print(Hola, ' + nombre + '. Tu nombre tiene ' + str(len(nombre)) +
'letras.')

sayHello('Alicia')
sayHello('Berto')
spam = 'Carolina'
sayHello(spam)
```

Si ejecutas este programa, verás algo así:

```
Hola, Alicia. Tu nombre tiene 6 letras.
Hola, Berto. Tu nombre tiene 5 letras.
Hola, Carolina. Tu nombre tiene 8 letras.
```

Cuando llamas a `decirHola()`, el argumento se asigna al parámetro `nombre`. Los parámetros son simplemente variables locales ordinarias. Como todas las variables locales, los valores en los parámetros serán olvidados cuando la llamada a la función retorne.

## Mostrando los Resultados del Juego

Volviendo al código fuente del juego:

```
20.     print('Te aproximas a la cueva...')
21.     time.sleep(2)
```

El módulo time tiene una función llamada `sleep()` que pone al programa en pausa. La línea 21 pasa el valor entero `2` de modo que `time.sleep()` pondrá al programa en pausa por 2 segundos.

```
22.     print('Es oscura y espeluznante...')
23.     time.sleep(2)
```

Aquí el código imprime algo más de texto y espera por otros 2 segundos. Estas pequeñas pausas agregan suspenso al juego, en lugar de mostrar todo el texto a la vez. En el programa Chistes del capítulo anterior, has llamado a la función `input()` para poner el juego en pausa hasta que el jugador pulsara la tecla INTRO. Aquí, el jugador no tiene que hacer nada excepto esperar un par de segundos.

```
24.     print('¡Un gran dragon aparece súbitamente frente a tí! Abre sus fauces
y...')
25.     print()
```

```
26.     time.sleep(2)
```

¿Qué ocurre a continuación? ¿Y cómo decide el programa? Esto se explica en la siguiente sección.

## Decidiendo Qué Cueva tiene el Dragón Amigable

```
28.     cuevaAmigable = random.randint(1, 2)
```

La línea 28 llama a la función `random.randint()` que devolverá `1` ó `2`. Este valor entero se almacena en `cuevaAmigable` y representa la cueva con el dragón amigable.

```
30.     if cuevaElegida == str(cuevaAmigable):
31.         print('¡Te regala su tesoro!')
```

La línea 30 comprueba si la cueva elegida por el jugador en la variable `cuevaElegida` (`'1'` or `'2'`) es igual a la cueva del dragón amistoso.

Pero el valor en `cuevaAmigable` es un entero porque `random.randint()` devuelve enteros. No puedes comparar cadenas y enteros con el signo `==`, porque **siempre** resultarán distintas. `'1'` no es igual a `1` y `'2'` no es igual a `2`.

Entonces se pasa `cuevaAmigable` a la función `str()`, la cual devuelve el valor de cadena de `cuevaAmigable`. De esta manera los valores serán el mismo tipo de datos y pueden ser comparados en forma relevante. También podríamos haber usado el siguiente código para convertir `cuevaElegida` a un valor entero:

```
    if int(cuevaElegida) == cuevaAmigable:
```

Si la condición es `True`, la línea 31 comunica al jugador que ha ganado el tesoro.

```
32.     else:
33.         print('¡Te engulle de un bocado!')
```

La línea 32 es una sentencia **else** (si no). La palabra reservada else siempre viene a continuación del bloque `if`. El bloque else se ejecuta si la condición de la sentencia if fue `False`. Piensa en esto como la forma del programa de decir, "Si esta condición es verdadera entonces ejecuta el bloque `if`, en caso contrario ejecuta el bloque `else`."

Recuerda colocar los dos puntos (el signo : ) luego de la palabra reservada `else`.

## Donde Comienza la Parte Principal

```
35. jugarDeNuevo = 'sí'
36. while jugarDeNuevo == 'sí' or jugarDeNuevo == 's':
```

La línea 35 es la primera línea que no es una sentencia `def` ni pertenece a un bloque `def`. Esta línea es donde la parte principal del programa comienza. Las sentencias def anteriores sólo definen las funciones, pero sin ejecutarlas.

Las líneas 35 y 36 configuran un bucle que contiene al resto del juego. Al final del juego, el jugador puede escribir si desea jugar de nuevo. Si es así, la ejecución vuelve a entrar al bucle `while` para ejecutar todo el juego otra vez. En caso contrario, la condición de la sentencia `while` será `False` y la ejecución continuará hasta el final del programa y terminará.

La primera vez que la ejecución llega a esta sentencia `while`, la línea 35 ha acabado de asignar `'sí'` a la variable `jugarDeNuevo`. Esto significa que la condición será `True`. De esta forma se garantiza que la ejecución entrará al bucle al menos una vez.

## *Llamando a las Funciones en el Programa*

```
38.     mostrarIntroducción()
```

La línea 38 llama a la función `mostrarIntroducción()`. Esta no es una función de Python, es la función que has definido anteriormente en la línea 4. Cuando se llama a esta función, la ejecución del programa salta a la primera línea en la función `mostrarIntroducción()` en la línea 5. Cuando todas las líneas en la función han sido ejecutadas, la ejecución vuelve a la línea 38 y continúa bajando.

```
40.     númeroDeCueva = elegirCueva()
```

La línea 40 también llama a una función que tú has definido. Recuerda que la función `elegirCueva()` permite al jugador elegir la cueva a la que desea entrar. Cuando se ejecuta `return cueva` en la línea 17, la ejecución del programa vuelve a la línea 40, y la llamada a `elegirCueva()` se evalúa al valor de retorno. Este valor de retorno es almacenado en una nueva variable llamada `númeroDeCueva`. Entonces la ejecución del programa continúa en la línea 42.

```
42.     explorarCueva(númeroDeCueva)
```

La línea 42 llama a tu función `explorarCueva()`, pasándole el valor en `númeroDeCueva` como argument. No sólo la ejecución salta a la línea 20, sino que el valor en `númeroDeCueva` se copia al parámetro `cuevaElegida` dentro de la función `explorarCueva()`. Esta es la función que mostrará

`'¡Te regala su tesoro!'` o `'¡Te engulle de un bocado!'` dependiendo de la cueva que el jugador elija.

### *Preguntando al Jugador si quiere Jugar de Nuevo*

```
44.     print('¿Quieres jugar de nuevo? (sí o no)')
45.     jugarDeNuevo = input()
```

Sin importar si el jugador gana o pierde, se le pregunta si quiere jugar de nuevo. La variable `jugarDeNuevo` almacena lo que haya ingresado el jugador. La línea 45 es la última línea del bloque `while`, de modo que el programa vuelve a la línea 36 para comprobar la condición del bucle `while`: `jugarDeNuevo == 'sí' or jugarDeNuevo == 's'`

Si el jugador ingresa la cadena `'sí'` o `'s'`, la ejecución entrará nuevamente al bucle en la línea 38.

Si el jugador ingresa `'no'` o `'n'`, o una tontería como `'Abraham Lincoln'`, entonces la condición será `False`. La ejecución del programa continúa a la línea a continuación del bloque `while`. Pero dado que no hay más líneas después del bloque `while`, el programa termina.

Una cosa a tener en cuenta: la cadena `'SÍ'` no es igual a la cadena `'sí'`. Si el jugador ingresa la cadena `'SÍ'`, entonces la condición de la sentencia `while` se evaluará a `False` y el programa terminará igualmente. Otros programas más adelante en este libro te mostrarán cómo evitar este problema.

¡Acabas de completar tu segundo juego! En Reino de Dragones, has usado mucho de cuanto aprendiste en el juego "Adivina el Número" y has aprendido unos cuantos trucos nuevos. Si no entendiste algunos de los conceptos en este programa, recorre cada línea del código fuente otra vez e intenta modificar el código fuente viendo cómo cambia el programa.

En el siguiente capítulo no crearás un juego, pero aprenderás cómo usar una funcionalidad de IDLE llamada depurador.

## Diseñando el Programa

Reino de Dragones es un juego simple. El resto de los juegos en este libro serán un poco más complicados. A veces ayuda escribir en papel todo lo que quieres que tu juego o programa haga antes de comenzar a escribir el código. Esto se llama "diseñar el programa".

Por ejemplo, puede ayudar dibujar un diagrama de flujo. Un **diagrama de flujo** es una ilustración que muestra cada posible acción que puede ocurrir en el juego, y qué acciones llevan a qué otras acciones. La Figura 6-2 es un diagrama de flujo para Reino de Dragones.

Para ver qué pasa en el juego, coloca tu dedo sobre el recuadro “Inicio”. Luego sigue una flecha desde ese recuadro hasta otro recuadro. Tu dedo es como la ejecución del programa. El programa termina cuando tu dedo llega al recuadro “Fin”.

Cuando llegas al recuadro “Comprobar dragón amistoso o hambriento”, puedes ir al recuadro “Jugador gana” o al recuadro “Jugador pierde”. Esta bifurcación muestra cómo el programa puede hacer diferentes cosas. De cualquier forma, ambos caminos conducirán al recuadro “Ofrece jugar de nuevo”.

## *Resumen*

En el juego “Reino de Dragones”, has creado tus propias funciones. Las funciones son un mini-programa dentro de tu programa. El código dentro de la función se ejecuta cuando la función es llamada. Al descomponer tu código en funciones, puedes organizar tu código en secciones más pequeñas y fáciles de entender.

Los argumentos son valores pasados al código de la función cuando la función es llamada. La propia llamada a la función se evalúa al valor de retorno.

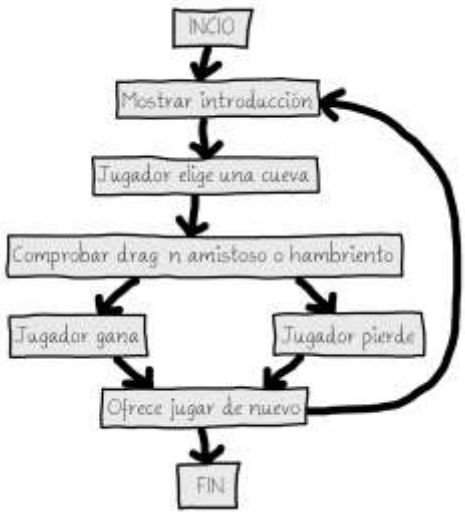



Figura 6-2: Diagrama de flujo para el juego Reino de Dragones.

También has aprendido acerca de entornos de variables. Las variables creadas dentro de una función existen en el entorno local, y las variables creadas fuera de todas las funciones existen en el entorno global. El código en el entorno global no puede usar las variables locales. Si una variable local tiene el mismo nombre que una variable en el entorno global, Python la considera una variables separada y asignar nuevos valores a la variable local no cambiará el valor de la variable global.

Los entornos de variables pueden parecer complicados, pero son útiles para organizar funciones como fragmentos de código separados del resto del programa. Dado que cada función tiene su propio entorno local, puedes estar seguro de que el código en una función no ocasionará errores en otras funciones.

Las funciones son tan útiles que casi todos los programas las usan. Entendiendo cómo funcionan las funciones, podemos ahorrarnos escribir muchas líneas de código y hacer que los errores sean más fáciles de arreglar.

# Capítulo 7
# USANDO EL DEPURADOR

Los tópicos cubiertos en este capítulo:
- 3 tipos diferentes de errores
- Depurador de IDLE
- Entrar en, sobre, salir
- Ir y salir
- Puntos de quiebre

## Bugs!

"En dos ocaciones me han preguntado 'Reza, Sr. Babbage ¿si pones en la máquina las figuras incorrectas, saldrán las respuestas correctas?' No consigo comprender correctamente el grado de confusión de ideas que puedan provocar dicha pregunta."

*-Charles Babbage, originador del concepto de una computadora programable, siglo 19.*

Si ingresas el código erróneo, la computadora no dará el programa correcto. Un programa de computadora siempre hará lo que tu le digas, pero lo que tu le digas al programa que haga puede que no sea lo que tu querías que haga. Estos errores son bugs de un programa de computadora. Los bugs ocurren cuando el programador no pensó cuidadosamente lo que el programa hace. Hay tres tipos de bugs que pueden ocurrir en tu programa:

- **Errores de Sintaxis**, estos provienen de errores de tipografía. Cuando el intérprete de Python vé un error de sintaxis, es porque tu código no se encuentra escrito correctamente en lenguaje Python. Un programa en Python aún con tan sólo un error de sintaxis no correrá.
- **Errores de Ejecución**, estos ocurren mientras el programa está corriendo. El programa funcionará hasta que alcanza la línea de código con el error y luego el programa terminará con un mensaje de error (eso se le llama **colapsar**, del ingés "crashing"). El intérprete mostrará un "traceback" (rastreo) y mostrará la línea donde ocurre el problema.
- **Errores de Semántica** son los más difíciles de solucionar. Estos errores no 'crashean' un programa, pero este no hará lo que el programador intencionaba. Por ejemplo, si el programador desea la variable `total` sea la suma de los valores en las variables `a`, `b` y `c` pero escribe `total = a * b * c`, entonces el valor en total será erróneo. Esto podría

colapsar el programa más adelante, pero no es inmediatamente obvio donde el error de semántica ocurre.

Hallar los bugs en un programa puede ser árduo ¡si es que siquiera los notas! Cuando corres tu programa, puedes descubrir que a veces ciertas funciones no son llamadas cuando deberían serlo, o tal vez son llamadas demasiadas veces. Puedes condicionar un ciclo while incorrectamente, ocacionando un número de ciclos incorrecto. (Un ciclo que nunca termina en tu programa es llamado **ciclo infinito**. Para parar este pgorama, puedes presionar **Ctrl-C** en la consola interactiva.) Cualquiera de estos pueden ocurrir accidentalmente en tu código si no eres cuidadoso.

De hecho, desde la consola interactiva, vé y crea un ciclo infinito al escribir el siguiente código (debes apretar INTRO dos veces para indicarle a la consola que has terminado de tipear el código del ciclo):

```
>>> while True:
...     print('¡¡¡Presiona Ctrl-C para parar este ciclo infinito!!!')
...
```

Ahora presione y mantenga la tecla Ctrl y presiona la tecla C para parar el programa. La consola interactiva se verá así:

```
¡¡¡Presiona Ctrl-C para parar este ciclo infinito!!!
¡¡¡Presiona Ctrl-C para parar este ciclo infinito!!!
¡¡¡Presiona Ctrl-C para parar este ciclo infinito!!!
¡¡¡Presiona Ctrl-C para parar este ciclo infinito!!!
¡¡¡Presiona Ctrl-C para parar este ciclo infinito!!!
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    while True: print('¡¡¡Presiona Ctrl-C para parar este ciclo infinito!!!')
KeyboardInterrupt
```

## El Depurador

Puede ser difícil darse cuenta cómo el código está causando un bug. Las líneas de código se ejecutan rápidamente y los valores en las variables cambian frecuentemente. Un **depurador** es un programa que te permite correr tu programa una línea de código a la vez en el mismo orden que Python. En depurador tambíen muestra en cada paso cuales son los valores almacenados en las variables.

### *Iniciando el Depurador*

Luego de abrir el archivo *dragón.py*, presona **Debug ► Debugger** para hacer aparecer el Debug Control (Control de Depuración) (Figura 7-1).

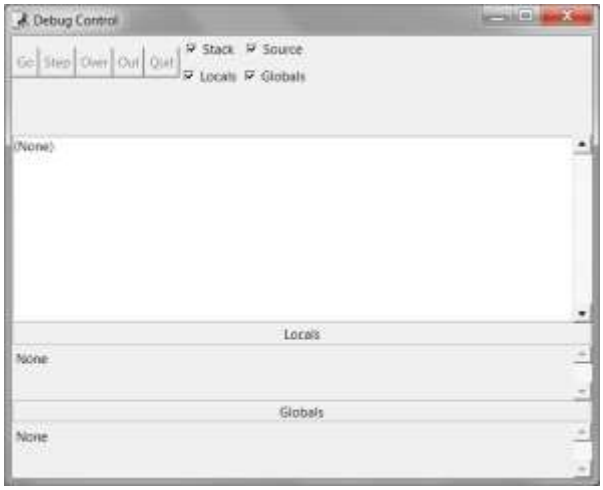



Figura 7-1: Ventana de Control de Depuración.

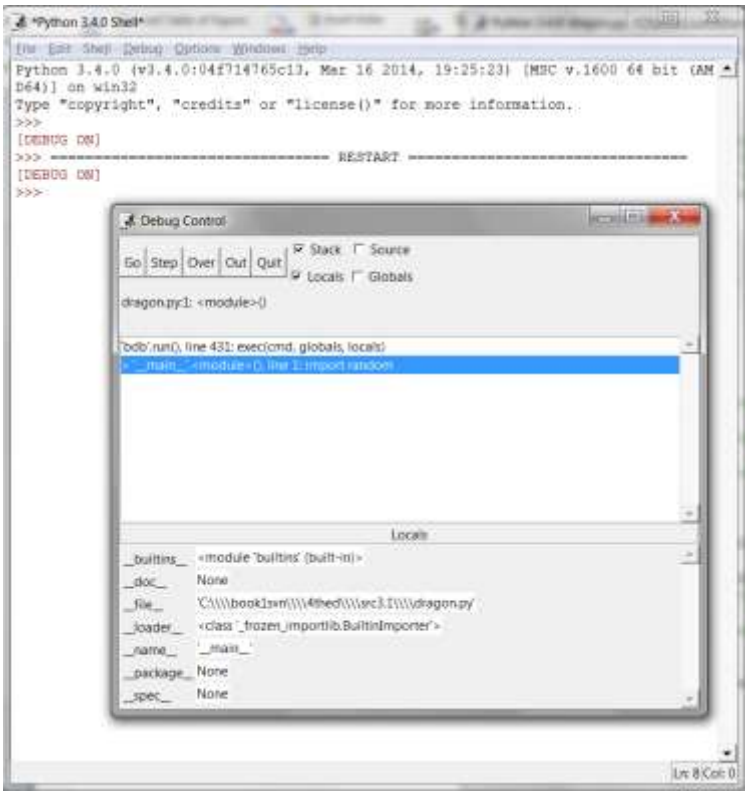



Figura 7-2: Coriendo Reino de Dragones bajo el depurador.

Ahora cuando corras el juego Reino de Dragones presionando F5, el depurador IDLE se activará. Esto es conocido como correr un programa "bajo un depurador". En la Ventana de Debug Control (Control de Depuración), selecciona los campos **Source** y **Globals**.

Cuando corres programas en Python con el depurador activado, el programa se frenará antes de ejecutar la primer línea de código. Si presionas sobre la barra del título del editor del archivo (y has seleccionado el campo **Source** en la ventala del Control de Depuración), la primera línea de codigo estará resaltada en gris. La ventana del Control de Depuración muestra que la ejecución se encuentra en la línea 1, la cuál es `import random`.

## Paso a Paso

El depurador te permite ejecutar una línea de código a la vez, llamado **paso a paso** (stepping en inglés). Para ejecutar una sola instrucción, presiona el botón **Step** en la ventana del Depurador. Vé y hazlo ahora. Python ejecutará la instrucción import random, y luego parará antes de ejecutar la próxima instrucción. La ventana de control ahora mostrará que la ejecucción ahora se encuentra en la línea 2, en import time. Presiona el botón **Quit** (Salir) para terminar el programa por ahora.

Aquí hay un resumen de lo que pasa cuando presionas el botón Step mientras corres el juego Reino de Dragones bajo el depurador. Presiona F5 para correr Reino de Dragones otra vez, luego sigue estas instrucciones:

1. Presiona el botón **Step** dos veces para ejecutar las dos líneas de `import`.
2. Presiona el botón **Step** otras tres veces para ejecutar las tres declaraciones `def`.
3. Presiona el botón **Step** otra vez para definir la variable `jugarDeNuevo`.
4. Presiona **Go** para correr el resto del programa, o presiona **Quit** para terminar el mismo.

La ventana del Control de Depuración mostrará que linea *está por ser* ejecutada cuando presiones Step. El depurador salteó la línea 3 debido a que es una línea en blanco. Notar que sólo se puede avanzar con el depurador, no puedes retroceder.

### *Área Globales*

El área de Globales en la ventana de control del depurador es donde se guardan todas las variables globales. Recuerda, las variables globales son aquellas creadas fuera de cualquier función (es decir, de alcalce global).

Debido a que las tres sentencias def ejecutan y definen funciones, apareceran en el area de globales.

Eĺ texto junto a los nombres de las funciones se verá como "<function checkCave at 0x012859B0>". Los nombres de módulos también tienen texto de aspecto confuso junto a ellos, tales como “<module 'random' from 'C:\\Python31\\lib\\random.pyc'>“. Esta información detallada es útil para los programadores avanzados, pero no necesitas saber que significa para depurar tus programas. Tan sólo con ver que las funciones y los módulos se encuentran en el área de globales te dirá que la función fue definida o el módulo importado.

También puedes ingorar las líneas `__builtins__`, `__doc__`, and `__name__`. (Son variables que aparecen en todo programa en Python.)

Cuando la variable `jugarDeNuevo` es creada, aparecerá en la sección Global. A su lado aparecerá el valor alojado en ella, el string `'si'`. El depurador te permite ver los valores de todas las variables en el programa mientras el mismo corre. Esto es útil para solucionar bugs en tu programa.

## *Área Locales*

Existe también un área Local, la cuál muestra el ámbito local de las variables y sus valores. El área local sólo tendrá variables cuando la ejecución del programa se encuentre dentro de una función. Cuando la ejecución se encuentre en el ámbito global, esta área estará en blanco.

## *Los botones Ir y Quitar (Go y Quit)*

Si te cansas de presionar el botón **Step** repetitivamente y solo quieres correr el programa normalmente, presiona el botón **Go** en la parte superior de la ventana de Control del Depurador. Esto le dirá al programa que corra normalmente en vez de paso a paso.

Para terminar el programa completamente, sólo presiona el botón **Quit** en la parte superior de la ventana de control. El programa terminará inmediatamente. Esto es útil si necesitas empezar a depurar de nuevo desde el comienzo del programa.

## *Entrar en, por encima, y salir*

Ejecuta el programa Reino de Dragones con el depurador. Ejecuta el programa paso a paso hasta que el depurador se encuentre en la línea 38. Como se muestra en la Figura 7-3, esta es la línea de la función `mostrarIntroduccion()`. El modo de paso a paso que has estado realizando se llama **Entrar En** (Stepping Into en ingles), porque el depurador entrará en la función cuando la misma es llamada. Esto es diferente a “Por Encima” (step over), que se explicará luego.

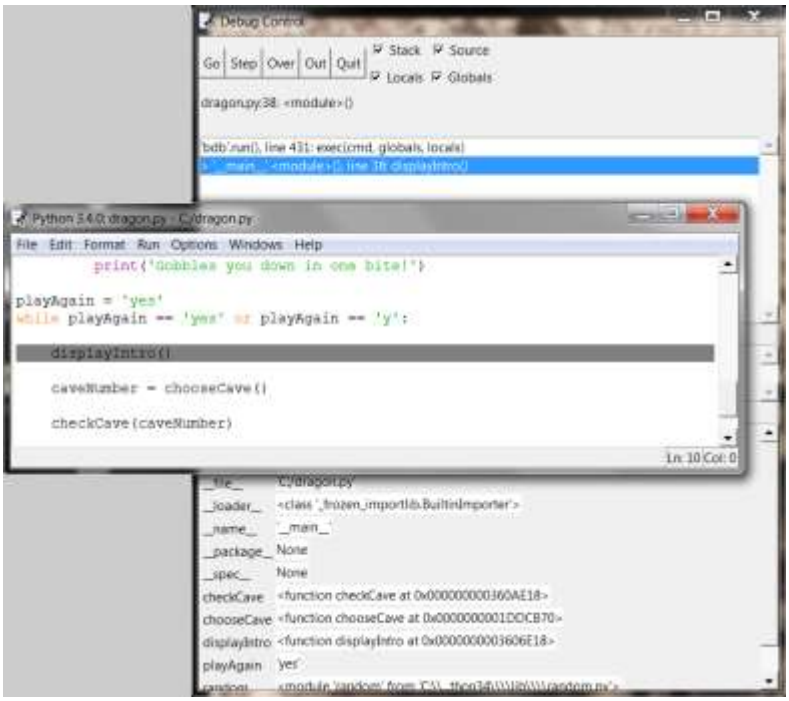



Figura 7-3: Continua el paso a paso hasta la línea 38.

Cuando la ejecución se pause en la línea 5, presionando **Step** una vez más se ingresará en la función `print()`. La función `print()` es una de las funciones incorporadas de Python, así que no es muy útil ingresar en ella con el depurador. Las funciones propias de Python como `print()`, `input()`, `str()`, o `random.randint()` ya fueron revisadas por errores. Puedes asumir que no son las partes causantes de bugs en tu programa.

Así que no quieres perder tiempo ingresando en el interior de la función `print()`. Entonces en vez de presionar **Step** para ingresar en el código de la función `print()`, presiona **Over**. Esto pasará por encima el código dentro de la función `print()`. El código dentro de `print()` será ejecutado a velocidad normal, y luego el depurador se pausará una vez que la ejecución vuelva de `print()`.

Pasar por encima es una manera conveniente de evitar pasar por código dentro de una función. El depurador ahora estará pausado en la línea 40, la línea con `númeroDeCueva = elegirCueva()`.

Presiona Step una vez mas para ingresar en la función `elegurCueva()`. Continua el paso a paso hasta la línea 15, la llamada a `input()`. El programa esperará hasta que ingreses una respuesta en la shell interactiva, tal como lo haría corriendo el programa normalmente. Si intentas presionando Step, nada pasará porque el programa esperará una respuesta del teclado.

Stepping over is a convenient way to skip stepping through code inside a function. The debugger will now be paused at line 40, `caveNumber = chooseCave()`.

Presiona **Step** una vez mas para ingresar en la función `elegurCueva()`. Continua el paso a paso hasta la línea 15, la llamada a `input()`. El programa esperará hasta que ingreses una respuesta en la shell interactiva, tal como lo haría corriendo el programa normalmente. Si intentas presionando Step, nada pasará porque el programa esperará una respuesta del teclado.

Vé a la consola interactiva y tipea cuál cueva deseas ingresar. El cursor parpadeante debe estar en la línea inferior en la consola interactva antes de que puedas tipear. Caso contrario el texto que ingreses no aparecerá.

Una vez que presiones INTRO, el depurador continuará el paso sobre las líneas. Presiona el botón **Out** en la ventana de control. A esto se le llama **Salir** (Stepping Out) porque hará que el depurador corra cuantas líneas sean necesarias hata salir de la función en la que se ecuentra. Luego de que sale, la ejecución debe estar en la línea siguiente a la línea que llamó la función.

Por ejemplo, al presionar **Out** dentro de la función `mostrarIntroducción()` en la línea 6, se correrá hasta que la función retorne a la línea posterior a la llamada a `mostrarIntroducción()`.

Si no te encuentras dentro de una función, presionar **Out** hará que el depurador ejecute todas las líneas restantes del programa. Este es el mismo comportamiento a presionar el botón **Go**.

Aquí un resumen de lo que cada botón hace:

- **Go** - Ejecuta el resto del código normalmente, o hasta que alcanza un punto de quiebre (break, que será descripto luego).
- **Step** - Ejecuta una línea de código. Si la línea es una llamada a una función, el depurador ingresará dentro de la función.
- **Over** - Ejecuta una línea de código. Si la línea es una llamada a una función, el depurador no ingresará dentro de la función.
- **Out** - Ejecuta líneas de código hasta que el depurador salga de la función en la que estaba cuando se presionó **Out**. Esto sale de la función.
- **Quit** - Termina el programa inmediatamente.

## Encuentra el Bug

El depurador puede ayudarte a encontrar la causa de bugs en tu programa. Por ejemplo, aquí hay un pequeño programa con un bug. El programa brinda un problema de suma aleatoria para que el usuario resuelva. En la consola interactiva, presiona en File, luego en New Window para abrir un nuevo editor de archivos. Tipea este programa en dicha ventana, y guarda el programa como *bugs.py*.

bugs.py

```
1. import random
2. numero1 = random.randint(1, 10)
3. numero2 = random.randint(1, 10)
4. print('¿Cuánto es ' + str(numero1) + ' + ' + str(numero2) + '?')
5. respuesta = input()
6. if respuesta == numero1 + numero2:
7.     print('¡Correcto!')
8. else:
9.     print('¡Nops! La respuesta es ' + str(numero1 + numero2))
```

Tipea el programa exáctamente como se muestra, incluso si ya sabes cuál es el bug. Luego intenta corer el programa presionando F5. Este es una simple pregunta aritmetica que te pide sumer dos números aleatorios. Aquí es lo que es posible que veas al correr el programa:

```
¿Cuánto es 5 + 1?
6
¡Nops! La respuesta es 6
```

¡Eso es un bug! El programa no colisiona pero no está trabajando correctamente. El programa dice que el usuario está equivocado incluso si ingresa la respuesta correcta.

Correr el programa en un depurador ayudará a encontrar la causa del bog. En la parte superior de la consola interactiva, presiona **Debug ► Debugger** para mostrar el control del depurador. En ella, selecciona las cuatro casillas (Stack, Source, Locals, y Globals). Esto hará que la ventana de control provea la mayor cantidad de información. Luego presiona F5 en la ventana del editor para correr el programa. Esta vez correra bajo el depurador.

```
1. import random
```

El depurador comenzará en la línea `import random`. Nada especial sucede aquí, así que presiona **Step** para ejecutarlo. Verás que el módulo `random` es agregado al área de globales (Globals).

```
2. numero1 = random.randint(1, 10)
```

Presiona **Step** otra vez para ejecutar la línea 2. Una nueva ventana de edición aparecera con el archivo *random.py* . Has ingresado dentro de la función `randint()` dentro del módulo random. Las funciones incorporadas en Python no serán fuente de tus errores, así que puedes presionar Out para salir de la función `randint()` y volver a tu programa. Luego cierra la ventana de *random.py*.

Click **Step** again to run line 2. A new file editor window will appear with the *random.py* file. You have stepped inside the `randint()` function inside the `random` module. Python's built-in functions won't be the source of your bugs, so click **Out** to step out of the `randint()` function and back to your program. Then close the *random.py* file's window.

```
3. numero2 = random.randint(1, 10)
```

La próxima vez, puedes presionar **Over** para saltar la función `randint()` en vez de ingresar en ella. La línea 3 también es una llamada a `randint()`. Evita ingresar en su código presionando **Over**.

```
4. print('What is ' + str(numero1) + ' + ' + str(numero2) + '?')
```

La línea 4 es una llamada a `print()` para mostrarle al jugador los números aleatorios. ¡Tu sabes que números el programa mostrará incluso antes de que los imprima! Tan sólo mira el área de globales en la ventana de contro. Puedes ver las variables `numero1` y `numero2`, y a su lado los valores enteros guardados en ellas.

La variable `numero1` posee el valor `4` y la variable `numero2` el valor `8`. Cuando presiones **Step**, el programa mostrará el string en la llamada `print()` con estosvalores. La función `str()` concatenará las versiones string de estos enteros. Cuando corrí el depurador, se vió como la Figura 7-4. (Tus valores aleatorios probablemente sean diferentes.)


Figura 7-4: numero1 establecido en 4 y numero2 en 8.

```
5. respuesta = input()
```

Presionando **Step** desde la línea 5 ejecutará `input()`. El depurador esperará hasta que el jugador ingrese una respuesta al programa. Ingresa la respuesta correcta (en mi caso, 19) en la consola interactiva. El depurador continuará y se moverá a la línea 6.

```
6. if respuesta == numero1 + numero2:
7.     print('Correct!')
```

La línea 6 es un condicional `if`. La condición es que el valor en la `respuesta` debe coincidir con la suma de `numero1` y `numero2`. Si la condicion es `True`, el depurador se moverá a la línea 7. Si es `False`, el depurador se moverá a la línea 9. Presiona **Step** una vez mas para descubrir adonde se moverá.

```
8. else:
9.     print('Nope! The answer is ' + str(numero1 + numero2))
```

¡El depurador ahora se encuentra en la línea 9! ¿Que sucedió? La condición en el if debe haber sido `False`. Mira los valores en `numero1`, `numero2`, y `respuesta`. Nota que `numero1` y `numero2` son enteros, así que su suma también debe ser un entero. Pero respuesta es una cadena.

Esto significa que `respuesta == numero1 + numero2` debio ser evaluado como `'12' == 12`. Una valor cadena y un valor entero siempre serán no iguales, así que la condición se evalua como `False`.

Este es el bug en el programa. El bug está en que usamos una respuesta cuando debimos usar `int(respuesta)`. Cambia la línea 6 para usar `int(respuesta) == numero1 + numero2` en vez de `respuesta == numero1 + numero2`, y corre el programa.

```
¿Cuanto es 2 + 3?
5
¡Correcto!
```

Esta vez, el programa funcionó correctamente. Córrelo una vez más e ingresa una respuesta errónea a propósito. Esto comprobará el programa completamente. ¡Ahora habrás depurado este programa! Recuerda, la computadora correrá tus programas exactamente como los tipeaste, incluso si lo que tipeaste no es lo que querias.

## Puntos de Quiebre

Ejecutar el código una línea a la vez puede ser demasiado lento. Con frecuencia quieres correr el programa normalmente hasta que alcance cierta línea. Un **punto quiebre** se establece en una línea donde quieres que el depurador tome el control una vez que la ejecución alcanzó dicha línea. Si crees que hay un programa en tu código, digamos, en la línea 17, tan sólo estableces un punto de quiebre en esa línea (o tal vez unas líneas atrás).

Cuando la ejecución alcance esa línea, el depurador "romperá hacia el depurador". Luego podrás correr las líneas una a la vez para ver que sucede. Presionar **Go** ejecutará el programa normalmente hasta que alcance otro punto quiebre o el final del programa.

Para establecer un punto quiebre, en el editor de texto haz click derecho sobre una línea y selecciona **Set Breakpoint** en el menú. Ahora el editor resaltará la línea en amarillo. Puedes establecer tantos puntos quiebre como desees. Para remover uno, clickea en la línea y selecciona **Clear Breakpoint** en el menú que aparece.

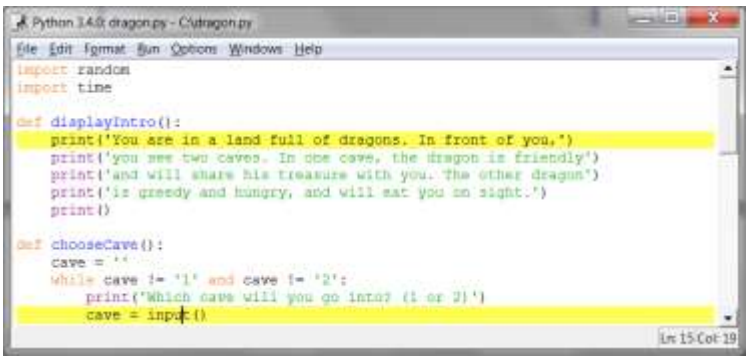



Figura 7-5: El editor con dos puntos quiebre establecidos.

## Ejemplos de Puntos Quiebre

Aquí una programa que simula lanzamientos de moneda llamando un `random.randint(0,1)`. La función al retornar `1` será "cara" y `0` será "cruz". La variable `lanzamientos` registará cuantos lanzamientos se efectuaron. La variable cara registará cuantos han salido cara.

El programa hará "lanzamientos de moneda" mil veces. Esto le tomaría a una persona más de una hora. ¡Pero la computadora puede hacerlo en un segundo! Escribe el siguiente código en el editor y guardalo como *lanzarMoneda.py*. También puedes descargar este código desde http://invpy.com/es/lanzarMoneda.py.

Si obtienes errores luego de escribir este código, compáralo con el código del libro con la herramienta online diff en http://invpy.com/es/diff/lanzarMoneda.

lanzarMoneda.py

```
 1. import random
 2. print('Lanzaré una moneda 1000 veces. Adivina cuantas veces caerá Cara.
(Presiona enter para comenzar)')
 3. input()
 4. lanzamientos = 0
 5. caras = 0
 6. while lanzamientos < 1000:
 7.     if random.randint(0, 1) == 1:
 8.         caras = caras + 1
 9.     lanzamientos = lanzamientos + 1
10.
11.     if lanzamientos == 900:
12.         print('900 lanzamientos y hubo ' + str(caras) + ' caras.')
13.     if lanzamientos == 100:
14.         print('En 100 lanzamientos, cara salió ' + str(caras) + ' veces.')
15.     if lanzamientos == 500:
16.         print('La mitad de los lanzamientos y cara salió ' + str(caras) + '
veces.')
17.
18. print()
19. print('De 1000 lanzamientos, al final cara salió ' + str(caras) + '
veces!')
20. print('¿Estuviste cerca?')
```

El programa corre bastante rápido. Toma más tiempo esperar a que el usuario presione INTRO que realizar los lanzamientos. Digamos que deseamos ver los lanzamientos de moneda uno a uno. En la consola interactiva, presiona **Debug ► Debugger** para abrir la ventana de control del depurador. Luego presiona F5 para correr el programa.

El programa comienza dentro del depurador en la línea 1. Presiona **Step** tres veces en la ventana de control para ejecutar las primeras tres líneas (estas son, líneas 1, 2 y 3). Notaras que los botones se deshabilitaran porque la función `input()` fue llamada y la consola interactiva está esperando al usuario. Clickea en la ventana de la consola y presiona INTRO. (Estate seguro de presionar debajo del texto en la consola interactiva, de lo contrario puede que IDLE no reciba tu tecla.)

Puedes presionar **Step** un par de veces mas, pero te encontarás que tardará un tiempo atravesar todo el programa. En vez, establece un punto de quiebre en las líneas 12, 14 y 16. El editor reseltará estas tres líneas como se muestra en la Figura 7-6.

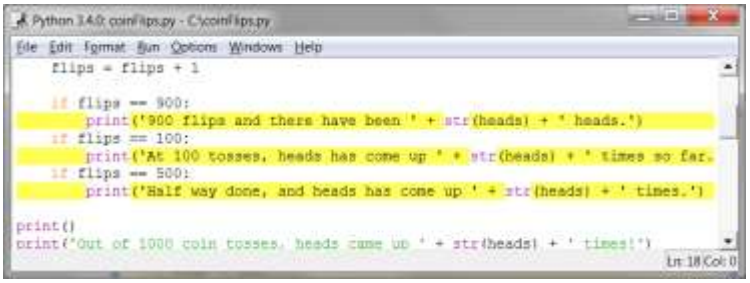



Figura 7-6: Tres puntos quiebre establecidos.

Luego de establecer los puntos quiebre, presiona **Go** en la ventana de control. El programa correrá a velocidad normal hasta toparse con el siguiente punto quiebre. Cuando `lanzamientos` se encuentra en `100`, el condicional del if en la línea 13 es `True`. Esto causa que la línea 14 (donde tenemos un break point) se ejecute, lo que le dice al depurador que frene el programa y tome el control. Mira la ventana de control del depurador en la sección de Globales para ver cuál es el valor de `lanzamientos` y `caras`.

Presiona nuevamente **Go** y el programa continuará hasta el siguiente punto quiebre en la línea 16. Otra vez, mira cómo los valores en `lanzamientos` y caras han cambiado.

Si presionas **Go** otra vez, la ejecución continuará hasta el último punto quiebre en la línea 12.

## *Resumen*

Escribir un programa es sólo la primer parte de programar. La siguiente parte es cerciorarse que lo escrito realmente funciona. Los depuradores te permiten atravesar el código una línea a la vez. Puedes examinar qué lineas se ejecutan en qué orden, y qué valores contienen las variables. Cuando esto es demasiado lento, puedes establecer puntos quiebres para frenar el depurador sólo en las línas que deseas.

Utilizar el depurador es una gran forma de entender exactamente lo que el programa está haciendo. Mientras que este libro explica todo el código dentro del mismo, el depurador puede ayudarte a encontrar más por tu cuenta.

# Capítulo 8
# DIAGRAMAS DE FLUJO

Temas Tratados En Este Capítulo:
- Cómo jugar al Ahorcado
- Arte ASCII
- Diseño de un Programa mediante Diagramas de Flujo.

En este capítulo, diseñarás el juego del Ahorcado. Este juego es más complicado que nuestro juego anterior, pero también más divertido. Como este juego es avanzado, deberías planearlo cuidadosamente antes creando un diagrama de flujo(explicado más adelante). En el siguiente capítulo, escribiremos el código para el Ahorcado.

## Cómo jugar al Ahorcado

El Ahorcado es un juego para dos personas, usualmente jugado con lápiz y papel. Un jugador piensa en una palabra y dibuja un espacio en la página por cada letra de la palabra. Entonces el segundo jugador intenta adivinar letras de la palabra.

Si adivina correctamente, el primer jugador escribe la letra en el espacio correspondiente. Si la letra no está en la palabra, el primer jugador dibuja una parte del cuerpo del ahorcado. Si el segundo jugador consigue adivinar todas las letras de la palabra antes de que el dibujo del ahorcado esté completo, gana. ¡Pero si no consigue adivinar la palabra a tiempo el segundo jugador pierde!

## Prueba de ejecución del Ahorcado

Aquí hay un ejemplo de lo que el jugador podría ver al ejecutar el programa Ahorcado que escribiremos en el próximo capítulo. El texto que el jugador ingresa se muestra en negrita.

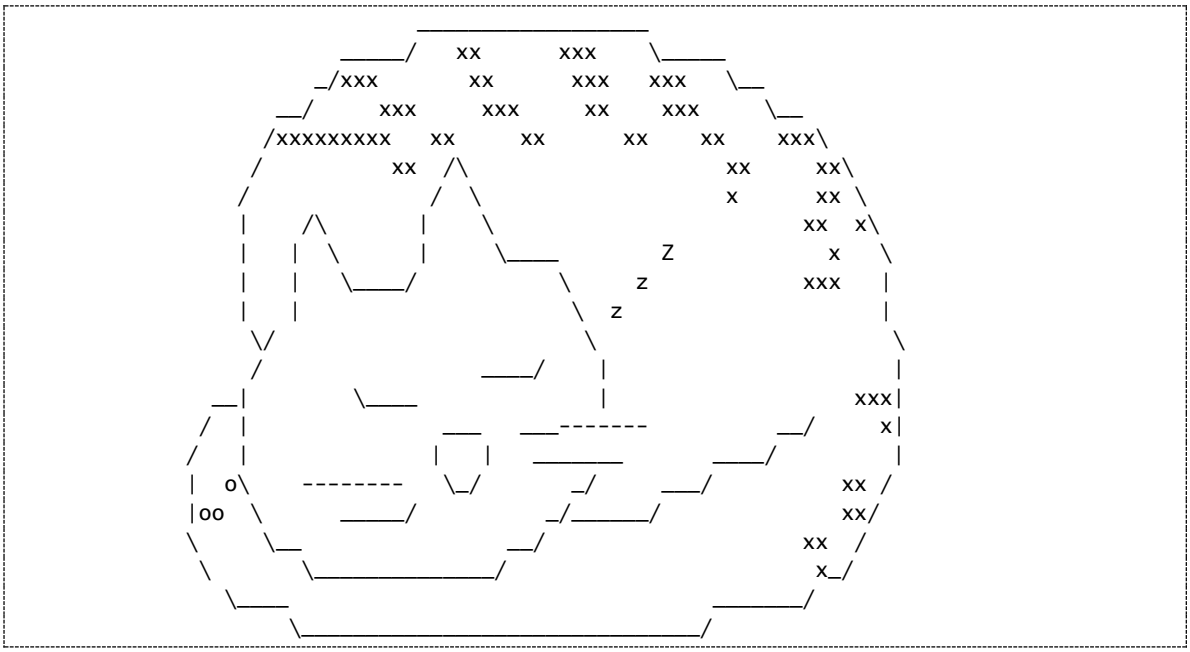
```
A H O R C A D O
  +---+
  |   |
      |
      |
      |
      |
=========
Letras incorrectas:
```

```
_ _ _ _
Adivina una letra.
o
  +---+
  |   |
      |
      |
      |
      |
=========
Letras incorrectas:
_ o _ o
Adivina una letra.
l
  +---+
  |   |
  O   |
      |
      |
      |
=========
Letras incorrectas: l
_ o _ o
Adivina una letra.
r
  +---+
  |   |
  O   |
  |   |
      |
      |
=========
Letras incorrectas: lr
_ o _ o
Adivina una letra.
n
  +---+
  |   |
  O   |
  |   |
      |
      |
=========
Letras incorrectas: lr
_ o n o
Adivina una letra.
o
```

```
Ya has probado esa letra. Elige otra.
Adivina una letra.
m
¡Sí! ¡La palabra secreta es "mono"! ¡Has ganado!
¿Quieres jugar de nuevo? (sí o no)
no
```

## Arte ASCII

Los gráficos para el Ahorcado son caracteres del teclado impresos en la pantalla. Este tipo de gráficos se llama **arte ASCII** (se pronuncia “asqui”), y fue una especie de precursor a emojii. Aquí hay un gato dibujado con arte ASCII:



## Diseño de un Programa mediante Diagramas de Flujo

Este juego es un poco más complicado que los que hemos visto hasta ahora, de modo que tómate un momento para pensar cómo está implementado. Primero necesitarás crear un diagrama de flujo (como el que hay al final del capítulo Reino de Dragones) para ayudar a visualizar lo que este programa hará. Este capítulo explicará lo que son los diagramas de flujo y por qué son útiles. El siguiente capítulo explicará el código fuente para el juego del Ahorcado.

Un **diagrama de flujo** es un diagrama que muestra una serie de pasos como recuadros conectados por flechas. Cada recuadro representa un paso, y las flechas muestran qué pasos llevan a qué otros pasos. Coloca tu dedo sobre el recuadro "Inicio" del diagrama de flujo y recorre el programa siguiendo las flechas a los otros recuadros hasta que llegues al recuadro “Fin”.

La Figura 8-1 es un diagrama de flujo completo para el Ahorcado. Sólo puedes moverte de un recuadro a otro en la dirección de la flecha. Nunca puedes volver hacia atrás a menos que haya una segunda flecha apuntando en dirección opuesta, como en el recuadro “El jugador ya ha probado esa letra”.

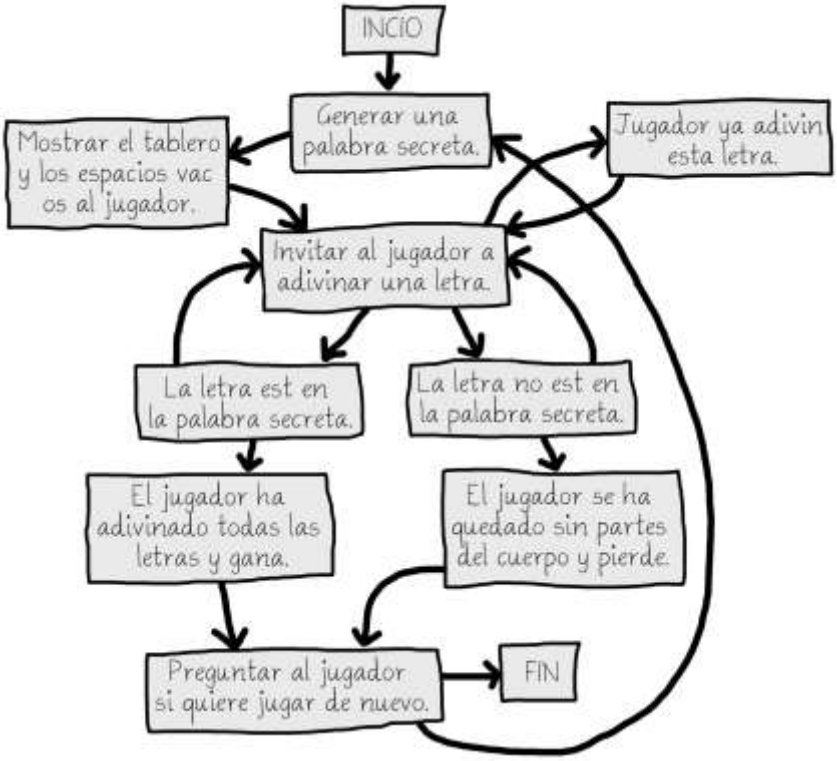



Figura 8-1: El diagrama de flujo completo del juego del Ahorcado.

Por supuesto, no es estrictamente *necesario* que hagas un diagrama de flujo. Podrías simplemente comenzar escribiendo código. Pero a menudo una vez que comiences a programar pensarás en cosas que es necesario agregar o cambiar. Podrías terminar teniendo que borrar una gran parte de tu código, lo que sería un desperdicio de esfuerzo. Para evitar esto, siempre es mejor planear cómo el programa va a funcionar antes de comenzar a escribirlo.

## Crear el Diagrama de Flujo

Tus diagramas de flujo no siempre tienen que verse exactamente como este. Siempre y cuando entiendas el diagrama de flujo que has hecho, será útil cuando comiences a escribir código. En la Figura 8-2 se muestra un diagrama de flujo que comienza con sólo un recuadro “Inicio” y un recuadro “Fin”:


Figura 8-2: Comienza tu diagrama de flujo con los recuadros Inicio y Fin.

Ahora piensa en lo que ocurre cuando juegas al Ahorcado. Primero, la computadora piensa en una palabra secreta. Luego el jugador intentará adivinar las letras. Agrega recuadros para estos eventos, como se muestra en la Figura 8-3. Los recuadros que son nuevos para cada diagrama de flujo se dibujan con bordes en línea quebrada.

Las flechas muestran el orden en que el programa debería moverse. Es decir, primero el programa debería generar una palabra secreta, y luego de eso debería invitar al jugador a adivinar una letra.


Figura 8-3: Dibuja los dos primeros pasos del Ahorcado como recuadros con descripciones.

Pero el juego no termina después de que el jugador prueba una letra. Necesita comprobar si esa letra pertenece o no a la palabra secreta.

## *Creando Ramificaciones a partir de un Recuadro del Diagrama de Flujo*

Hay dos posibilidades: la letra puede estar en la palabra o no. Esto significa que necesitas agregar dos nuevos recuadros al diagrama de flujo, uno por cada caso. Esto crea una ramificación en el diagrama de flujo, como se muestra en la Figura 8-4:


Figura 8-4: La rama consiste en dos flechas hacia recuadros diferentes.

Si la letra está en la pantalla secreta, comprueba si el jugador ha adivinado todas las letras y ha ganado. Si la letra no está en la palabra del juego, se agrega otra parte del cuerpo al ahorcado. Agrega recuadros para esos casos también.

No **necesitas** una flecha desde el casillero "La letra está en la palabra secreta" al casillero "El jugador se ha quedado sin partes del cuerpo y pierde", porque es imposible perder mientras el jugador acierte. También es imposible ganar mientras el jugador no acierte, de modo que tampoco precisamos dibujar esa otra flecha. El diagrama de flujo se ve ahora como la Figura 8-5.

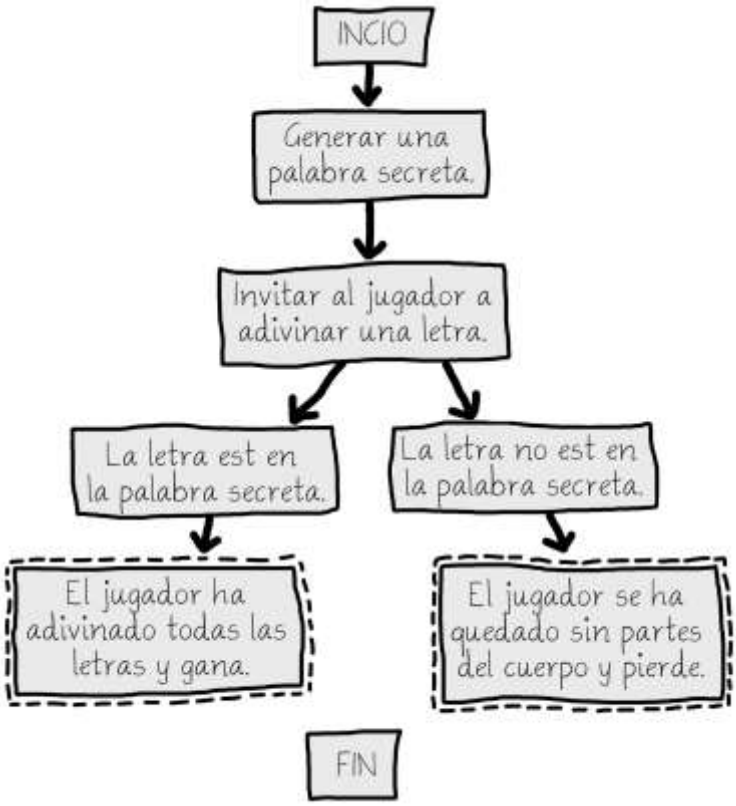



Figura 8-5: Luego de la ramificación, los pasos continúan por caminos separados..

## *Finalizar o Reiniciar el Juego*

Una vez que el jugador ha ganado o perdido, pregúntale si desea jugar de nuevo con una nueva palabra secreta. Si el jugador no quiere jugar de nuevo, el programa termina. Si el programa no termina, pensamos una nueva palabra secreta. Esto se muestra en la Figura 8-6.


Figura 8-6: El diagrama de flujo se ramifica al preguntar al jugador si quiere jugar de nuevo.

## *Adivinando Nuevamente*

El jugador no adivina una letra sólo una vez. Tiene que continuar probando letras hasta que gane o pierda. Necesitarás dibujar dos nuevas flechas, como se muestra en la Figura 8-7.


Figura 8-7: Las nuevas flechas (resaltadas) denotan que el jugador puede adivinar otra vez.

¿Qué ocurre si el jugador prueba la misma letra más de una vez? En lugar de ganar o perder en este caso, le permitiremos probar una nueva letra. Este nuevo recuadro se muestra en la Figura 8-8.

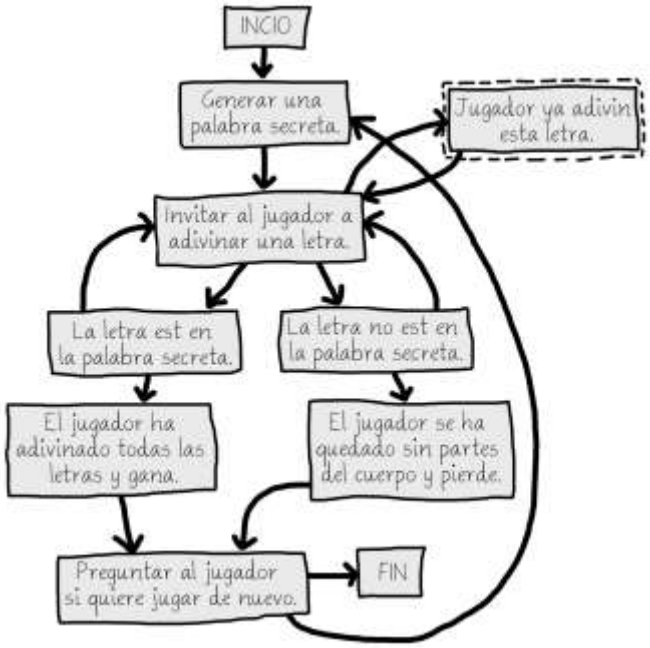



Figura 8-8: Agregamos un paso en caso de que el jugador pruebe una letra por segunda vez.

## *Ofreciendo Retroalimentación al Jugador*

El jugador necesita saber qué está pasando en el juego. El programa debería mostrar el tablero del Ahorcado y la palabra secreta (con espacios en blanco en las letras que aún no ha adivinado). Estas ayudas visuales permitirán que el jugador sepa qué tan cerca está de ganar o perder el juego.

Esta información deberá ser actualizada cada vez que el jugador pruebe una letra. Agrega un recuadro “Mostrar el tablero y los espacios vacíos al jugador” al diagrama de flujo entre los recuadros “Generar una palabra secreta” e “Invitar al jugador a adivinar una letra”. Estos recuadros se muestran en la Figura 8-9.

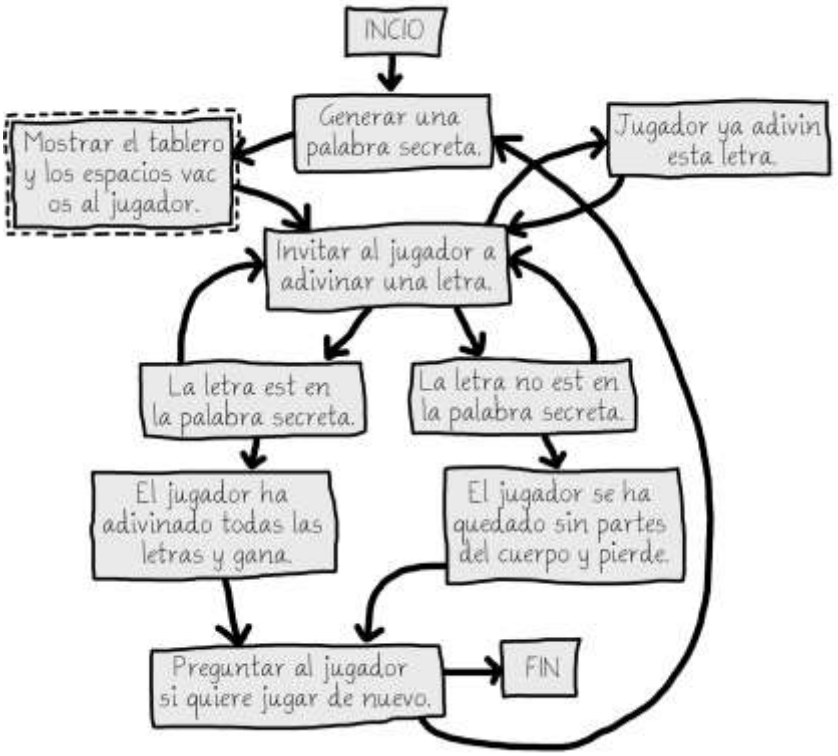



Figura 8-9: Agregamos “Mostrar el tablero y los espacios vacíos al jugador” para dar información al jugador.

¡Eso se ve bien! Este diagrama de flujo reproduce completamente todo lo que puede ocurrir en el Ahorcado y en qué orden. Cuando diseñes tus propios juegos, un diagrama de flujo puede ayudarte a recordar todo lo que necesitas codificar.

## Resumen

Puede parecer muchísimo trabajo dibujar un diagrama de flujo del programa primero. Después de todo, ¡la gente quiere jugar juegos, no mirar diagramas de flujo! Pero es mucho más fácil hacer cambios y notar problemas pensando un poco sobre cómo funciona el programa antes de escribir el código.

Si te lanzas a escribir el código primero, puedes llegar a descubrir problemas que requieren que cambies el código que has escrito. Cada vez que cambias tu código, estás corriendo el riesgo de introducir nuevos errores por cambiar demasiado, o demasiado poco. Es mucho mejor saber qué es lo que quieres construir antes de construirlo.

# Capítulo 10
# TA TE TI

Temas Tratados En Este Capítulo:
- Inteligencia Artificial
- Referencias en Listas
- Evaluación en Cortocircuito
- El Valor `None`

Este capítulo presenta un juego de Ta Te Ti contra una inteligencia artificial simple. Una **inteligencia artificial** (**IA**) es un programa de computadora que puede responder inteligentemente a los movimientos del jugador. Este juego no introduce ningún nuevo concepto que sea complicado. La inteligencia artificial del juego de Ta Te Ti consiste en sólo unas pocas líneas de código.

Dos personas pueden jugar Ta Te Ti con lápiz y papel. Un jugador es X y el otro es O. En un tablero consistente en nueve cuadrados, los jugadores toman turnos para colocar sus X u O. Si un jugador consigue ubicar tres de sus marcas en el tablero sobre la misma línea, columna o alguna de las dos diagonales, gana. Cuando el tablero se llena y ningún jugador ha ganado, el juego termina en empate.

Este capítulo no introduce muchos nuevos conceptos de programación. Hace uso de nuestro conocimiento adquirido hasta ahora para crear un jugador inteligente de Ta Te Ti. Empecemos mirando una prueba de ejecución del programa. El jugador hace su movimiento escribiendo el número del espacio en el que quiere jugar. Estos números están dispuestos de igual forma que las teclas numéricas en tu teclado (ver Figura 10-2).

## Prueba de Ejecución de Ta Te Ti

```
¡Bienvenido al Ta Te Ti!
¿Deseas ser X o O?
X
La computadora irá primero.
   |   |
 O |   |
   |   |
-----------
   |   |
   |   |
```

```
   |   |
-----------
   |   |
   |   |
   |   |
¿Cuál es tu próxima jugada? (1-9)
3
   |   |
 O |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
 O |   | X
   |   |
¿Cuál es tu próxima jugada? (1-9)
4
   |   |
 O |   | O
   |   |
-----------
   |   |
 X |   |
   |   |
-----------
   |   |
 O |   | X
   |   |
¿Cuál es tu próxima jugada? (1-9)
5
   |   |
 O | O | O
   |   |
-----------
   |   |
 X | X |
   |   |
-----------
   |   |
 O |   | X
   |   |
¡La computadora te ha vencido! Has perdido.
¿Deseas volver a jugar? (sí/no)?
no
```

## Código Fuente del Ta Te Ti

En una nueva ventana del editor de archivos, escribe el siguiente código y guárdalo como tateti.py. Luego ejecuta el juego pulsando F5.

tateti.py

```
 1. # Ta Te Ti
 2.
 3. import random
 4.
 5. def dibujarTablero(tablero):
 6.     # Esta función dibuja el tablero recibido como argumento.
 7.
 8.     # "tablero" es una lista de 10 strings representando la pizarra
(ignora índice 0)
 9.     print('   |   |')
10.     print(' ' + tablero[7] + ' | ' + tablero[8] + ' | ' + tablero[9])
11.     print('   |   |')
12.     print('-----------')
13.     print('   |   |')
14.     print(' ' + tablero[4] + ' | ' + tablero[5] + ' | ' + tablero[6])
15.     print('   |   |')
16.     print('-----------')
17.     print('   |   |')
18.     print(' ' + tablero[1] + ' | ' + tablero[2] + ' | ' + tablero[3])
19.     print('   |   |')
20.
21. def ingresaLetraJugador():
22.     # Permite al jugador typear que letra desea ser.
23.     # Devuelve una lista con las letras de los jugadores como primer item,
y la de la computadora como segundo.
24.     letra = ''
25.     while not (letra == 'X' or letra == 'O'):
26.         print('¿Deseas ser X o O?')
27.         letra = input().upper()
28.
29.     # el primer elemento de la lista es la letra del jugador, el segundo
es la letra de la computadora.
30.     if letra == 'X':
31.         return ['X', 'O']
32.     else:
33.         return ['O', 'X']
34.
35. def quienComienza():
36.     # Elije al azar que jugador comienza.
37.     if random.randint(0, 1) == 0:
```

```
 38.         return 'La computadora'
 39.     else:
 40.         return 'El jugador'
 41.
 42. def jugarDeNuevo():
 43.     # Esta funcion devuelve True (Verdadero) si el jugador desea volver a
jugar, de lo contrario devuelve False (Falso).
 44.     print('¿Deseas volver a jugar? (sí/no)?')
 45.     return input().lower().startswith('s')
 46.
 47. def hacerJugada(tablero, letra, jugada):
 48.     tablero[jugada] = letra
 49.
 50. def esGanador(ta, le):
 51.     # Dado un tablero y la letra de un jugador, devuelve True (verdadero)
si el mismo ha ganado.
 52.     # Utilizamos reemplazamos tablero por ta y letra por le para no
escribir tanto.
 53.     return ((ta[7] == le and ta[8] == le and ta[9] == le) or # horizontal
superior
 54.     (ta[4] == le and ta[5] == le and ta[6] == le) or # horizontal medio
 55.     (ta[1] == le and ta[2] == le and ta[3] == le) or # horizontal inferior
 56.     (ta[7] == le and ta[4] == le and ta[1] == le) or # vertical izquierda
 57.     (ta[8] == le and ta[5] == le and ta[2] == le) or # vertical medio
 58.     (ta[9] == le and ta[6] == le and ta[3] == le) or # vertical derecha
 59.     (ta[7] == le and ta[5] == le and ta[3] == le) or # diagonal
 60.     (ta[9] == le and ta[5] == le and ta[1] == le)) # diagonal
 61.
 62. def obtenerDuplicadoTablero(tablero):
 63.     # Duplica la lista del tablero y devuelve el duplicado.
 64.     dupTablero = []
 65.
 66.     for i in tablero:
 67.         dupTablero.append(i)
 68.
 69.     return dupTablero
 70.
 71. def hayEspacioLibre(tablero, jugada):
 72.     # Devuelte true si hay espacio para efectuar la jugada en el tablero.
 73.     return tablero[jugada] == ' '
 74.
 75. def obtenerJugadaJugador(tablero):
 76.     # Permite al jugador escribir su jugada.
 77.     jugada = ' '
 78.     while jugada not in '1 2 3 4 5 6 7 8 9'.split() or not
hayEspacioLibre(tablero, int(jugada)):
 79.         print('¿Cuál es tu próxima jugada? (1-9)')
```

```
 80.         jugada = input()
 81.     return int(jugada)
 82.
 83. def elegirAzarDeLista(tablero, listaJugada):
 84.     # Devuelve una jugada válida en el tablero de la lista recibida.
 85.     # Devuelve None si no hay ninguna jugada válida.
 86.     jugadasPosibles = []
 87.     for i in listaJugada:
 88.         if hayEspacioLibre(tablero, i):
 89.             jugadasPosibles.append(i)
 90.
 91.     if len(jugadasPosibles) != 0:
 92.         return random.choice(jugadasPosibles)
 93.     else:
 94.         return None
 95.
 96. def obtenerJugadaComputadora(tablero, letraComputadora):
 97.     # Dado un tablero y la letra de la computadora, determina que jugada
efectuar.
 98.     if letraComputadora == 'X':
 99.         letraJugador = 'O'
100.     else:
101.         letraJugador = 'X'
102.
103.     # Aquí está nuestro algoritmo para nuestra IA (Inteligencia Artifical)
del Ta Te Ti.
104.     # Primero, verifica si podemos ganar en la próxima jugada
105.     for i in range(1, 10):
106.         copia = obtenerDuplicadoTablero(tablero)
107.         if hayEspacioLibre(copia, i):
108.             hacerJugada(copia, letraComputadora, i)
109.             if esGanador(copia, letraComputadora):
110.                 return i
111.
112.     # Verifica si el jugador podría ganar en su próxima jugada, y lo
bloquea.
113.     for i in range(1, 10):
114.         copia = obtenerDuplicadoTablero(tablero)
115.         if hayEspacioLibre(copia, i):
116.             hacerJugada(copia, letraJugador, i)
117.             if esGanador(copia, letraJugador):
118.                 return i
119.
120.     # Intenta ocupar una de las esquinas de estar libre.
121.     jugada = elegirAzarDeLista(tablero, [1, 3, 7, 9])
122.     if jugada != None:
123.         return jugada
```

```
124.
125.     # De estar libre, intenta ocupar el centro.
126.     if hayEspacioLibre(tablero, 5):
127.         return 5
128.
129.     # Ocupa alguno de los lados.
130.     return elegirAzarDeLista(tablero, [2, 4, 6, 8])
131.
132. def tableroCompleto(tablero):
133.     # Devuelve True si cada espacio del tablero fue ocupado, caso
contrario devuele False.
134.     for i in range(1, 10):
135.         if hayEspacioLibre(tablero, i):
136.             return False
137.     return True
138.
139.
140. print('¡Bienvenido al Ta Te Ti!')
141.
142. while True:
143.     # Resetea el tablero
144.     elTablero = [' '] * 10
145.     letraJugador, letraComputadora = ingresaLetraJugador()
146.     turno = quienComienza()
147.     print(turno + ' irá primero.')
148.     juegoEnCurso = True
149.
150.     while juegoEnCurso:
151.         if turno == 'El jugador':
152.             # Turno del jugador
153.             dibujarTablero(elTablero)
154.             jugada = obtenerJugadaJugador(elTablero)
155.             hacerJugada(elTablero, letraJugador, jugada)
156.
157.             if esGanador(elTablero, letraJugador):
158.                 dibujarTablero(elTablero)
159.                 print('¡Felicidades, has ganado!')
160.                 juegoEnCurso = False
161.             else:
162.                 if tableroCompleto(elTablero):
163.                     dibujarTablero(elTablero)
164.                     print('¡Es un empate!')
165.                     break
166.                 else:
167.                     turno = 'La computadora'
168.
169.         else:
```

```
170.             # Turno de la computadora
171.             jugada = obtenerJugadaComputadora(elTablero, letraComputadora)
172.             hacerJugada(elTablero, letraComputadora, jugada)
173.
174.             if esGanador(elTablero, letraComputadora):
175.                 dibujarTablero(elTablero)
176.                 print('¡La computadora te ha vencido! Has perdido.')
177.                 juegoEnCurso = False
178.             else:
179.                 if tableroCompleto(elTablero):
180.                     dibujarTablero(elTablero)
181.                     print('¡Es un empate!')
182.                     break
183.                 else:
184.                     turno = 'El jugador'
185.
186.     if not jugarDeNuevo():
187.         break
```

## Diseñando el Programa

La Figura 10-1 muestra cómo se vería un diagrama de flujo del Ta Te Ti. En nuestro programa del Ta Te Ti el jugador elige si quiere ser X u O. Quién toma el primer turno se elige al azar. Luego el jugador y la computadora toman turnos para jugar.

Los recuadros a la izquierda del diagrama de flujo son lo que ocurre durante el turno del jugador. El lado derecho muestra lo que ocurre durante el turno de la computadora. El jugador tiene un recuadro extra para dibujar el tablero ya que la computadora no precisa ver el tablero impreso en la pantalla. Luego de que el jugador o la computadora hacen su movimiento, revisamos si han ganado u ocasionado un empate, y entonces cambia el turno del juego. Después de que termina el juego, le preguntamos al jugador si desea jugar otra vez.

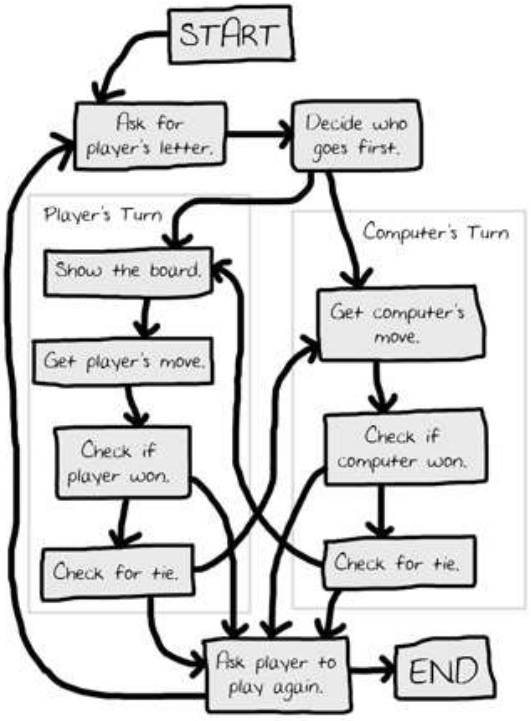



Figura 10-1: Diagrama de flujo para el Ta Te Ti

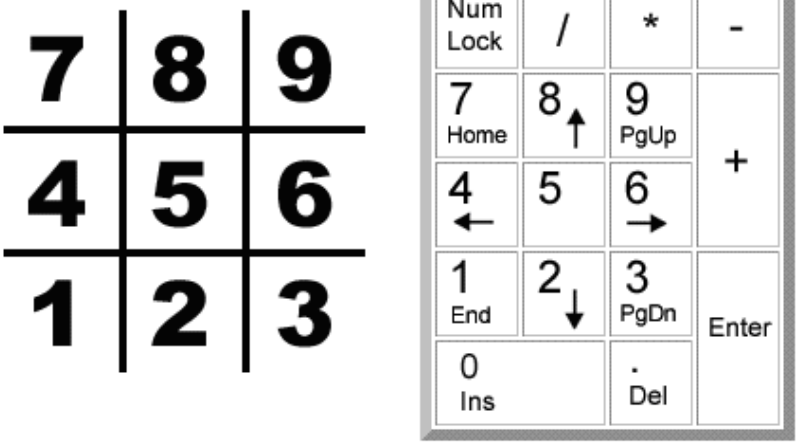



Figura 10-2: El tablero está ordenado igual que el teclado numérico de la computadora.

### *Representando el Tablero como Datos*

Primero, necesitamos entender cómo vamos a representar el tablero como una variable. Sobre papel, el tablero de Ta Te Ti se dibuja como un par de líneas horizontales y un par de líneas verticales, con una X, una O o un espacio vacío en cada una de las nueve regiones formadas.

En el programa, el tablero de Ta Te Ti se representa como una lista de cadenas. Cada cadena representa uno de los nueve espacios en el tablero. Para que sea más fácil recordar qué índice de la lista corresponde a cada espacio, los ordenaremos igual que en el tablero numérico del teclado, como se muestra en la Figura 10-2.

Las cadenas serán `'X'` para el jugador X, `'O'` para el jugador O, o un espacio simple `' '` para un espacio vacío.

Entonces si una lista de diez cadenas se guardase en una variable llamada tablero, `tablero[7]` sería el espacio superior izquierdo en el tablero. De la misma forma `tablero[5]` sería el centro, `tablero[4]` sería el costado izquierdo, etcétera. El programa ignorará la cadena en el índice `0` de la lista. El jugador entrará un número de `1` a `9` para decirle al juego sobre qué espacio quiere jugar.

## IA del Juego

La IA necesitará poder ver el tablero y decidir sobre qué tipo de espacios mover. Para ser claros, definiremos tres tipos de espacios en el tablero de Ta Te Ti: esquinas, lados y el centro. La Figura 10-3 presenta un esquema de qué es cada espacio.

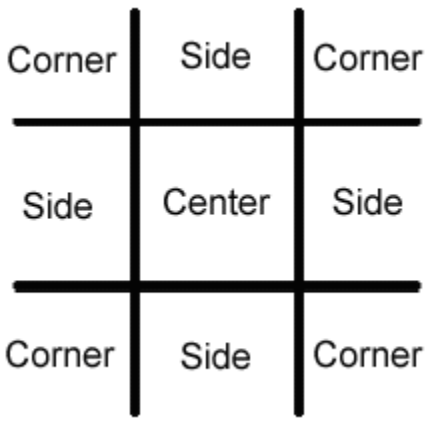



Figura 10-3: Ubicación de los lados, esquinas y centro en el tablero.

La astucia de la IA para jugar al Ta Te Ti seguirá un algoritmo simple. Un **algoritmo** es una serie finita de instrucciones para computar un resultado. Un único programa puede hacer uso de varios algoritmos diferentes. Un algoritmo puede representarse con un diagrama de flujo. El algoritmo de la IA del Ta Te Ti computa la mejor movida disponible, como se muestra en la Figura 10-4.

El algoritmo de la IA consiste en los siguientes pasos:

1. Primero, ver si hay un movimiento con el que la computadora pueda ganar el juego. Si lo hay, hacer ese movimiento. En caso contrario, ir al paso 2.
2. Ver si existe un movimiento disponible para el jugador que pueda hacer que la computadora pierda el juego. Si existe, la computadora debería jugar en ese lugar para bloquear la jugada ganadora. En caso contrario, ir al paso 3.
3. Comprobar si alguna de las esquinas (espacios 1, 3, 7, ó 9) está disponible. Si lo está, mover allí. Si no hay ninguna esquina disponible, ir al paso 4.
4. Comprobar si el centro está libre. Si lo está, jugar en el centro. Si no lo está, ir al paso 5.
5. Jugar en cualquiera de los lados (espacios 2, 4, 6, u 8). No hay más pasos, ya que si hemos llegado al paso 5 los únicos espacios restantes son los lados.

Todo esto ocurre dentro del casillero "Obtener movimiento de la computadora." en nuestro diagrama de flujo de la Figura 10-1. Podrías añadir esta información al diagrama de flujo con los recuadros de la Figura 10-4.

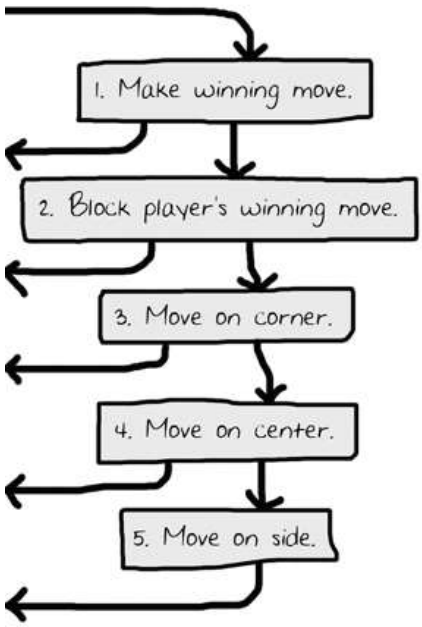



Figura 10-4: Los cinco pasos del algoritmo "Obtener movimiento de la computadora". Las flechas salientes van al recuadro "Comprobar si la computadora ha ganado".

Este algoritmo es implementado en la función `obtenerJugadaComputadora()` y las otras funciones llamadas por `obtenerJugadaComputadora()`.

## The Start of the Program

```
 1. # Ta Te Ti
 2.
 3. import random
```

El primer par de líneas son un comentario y la importación del módulo random para poder llamar a la función `randint()`.

## Dibujando el Tablero en la Pantalla

```
 5. def dibujarTablero(tablero):
 6.     # Esta función dibuja el tablero recibido como argumento.
 7.
 8.     # "tablero" es una lista de 10 strings representando la pizarra
(ignora índice 0)
 9.     print('   |   |')
10.     print(' ' + tablero[7] + ' | ' + tablero[8] + ' | ' + tablero[9])
11.     print('   |   |')
12.     print('-----------')
13.     print('   |   |')
14.     print(' ' + tablero[4] + ' | ' + tablero[5] + ' | ' + tablero[6])
15.     print('   |   |')
16.     print('-----------')
17.     print('   |   |')
18.     print(' ' + tablero[1] + ' | ' + tablero[2] + ' | ' + tablero[3])
19.     print('   |   |')
```

La función `dibujarTablero()` imprimirá el tablero de juego representado por el parámetro tablero. Recuerda que nuestro tablero se representa como una lista de diez cadenas, donde la cadena correspondiente al índice `1` es la marca en el espacio 1 sobre el tablero del Ta Te Ti. La cadena en el índice `0` es ignorada. Muchas de nuestras funciones operarán pasando una lista de diez cadenas a modo de tablero.

Asegúrate de escribir correctamente los espacios en las cadenas, ya que de otra forma el tablero se verá raro al imprimirse en pantalla. Aquí hay algunas llamadas de ejemplo (con un argumento como tablero) a `dibujarTablero()` junto con las correspondientes salidas de la función:

```
>>> drawBoard([' ', ' ', ' ', ' ', 'X', 'O', ' ', 'X', ' ', 'O']
   |   |
 X |   | O
```

```
   |   |
-----------
   |   |
 X | O |
   |   |
-----------
   |   |
   |   |
   |   |
>>> [' ', 'O', 'O', ' ', ' ', 'X', ' ', ' ', ' ', ' ']
   |   |
   |   |
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
 O | O |
   |   |
>>> [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
```

## *Dejando al Jugador elegir X u O*

```
 21. def ingresaLetraJugador():
 22.     # Permite al jugador typear que letra desea ser.
 23.     # Devuelve una lista con las letras de los jugadores como primer item,
y la de la computadora como segundo.
 24.     letra = ''
 25.     while not (letra == 'X' or letra == 'O'):
 26.         print('¿Deseas ser X o O?')
 27.         letra = input().upper()
```

La función `ingresaLetraJugador()` pregunta al jugador si desea ser X u O. Continuará preguntando al jugador hasta que este escriba X u O. La línea 27 cambia automáticamente la cadena devuelta por la llamada a `input()` a letras mayúsculas con el método de cadena `upper()`.

La condición del bucle while contiene paréntesis, lo que significa que la expresión dentro del paréntesis es evaluada primero. Si se asignara `'X'` a la variable `letra`, la expresión se evaluaría de esta forma:

```
not (letra == 'X' or letra == 'O')
      ▼
not ('X' == 'X'    or 'X' == 'O')
      ▼
not (   True       or    False)
      ▼
not (True)
      ▼
not True
      ▼
    False
```

Si letra tiene valor `'X'` o `'O'`, entonces la condición del bucle es `False` y permite que la ejecución del programa continúe luego del bloque `while`.

```
 29.     # el primer elemento de la lista es la letra del jugador, el segundo
es la letra de la computadora.
 30.     if letra == 'X':
 31.         return ['X', 'O']
 32.     else:
 33.         return ['O', 'X']
```

Esta función devuelve una lista con dos elementos. El primer elemento (la cadena del índice `0`) será la letra del jugador, y el segundo elemento (la cadena del índice `1`) será la letra de la computadora. Estas sentencias `if-else` elige la lista adecuada a devolver.

## *Decidiendo Quién Comienza*

```
 35. def quienComienza():
 36.     # Elije al azar que jugador comienza.
 37.     if random.randint(0, 1) == 0:
 38.         return 'La computadora'
 39.     else:
 40.         return 'El jugador'
```

La función `quienComienza()` lanza una moneda virtual para determinar quien comienza entre la computadora y el jugador. El lanzamiento virtual de moneda se realiza llamando a `random.randint(0, 1)`. Si esta llamada a función devuelve `0`, la función `quienComienza()` devuelve la cadena `'La computadora'`. De lo contrario, la función devuelve la cadena `'El jugador'`. El código que llama a esta función usará el valor de retorno para saber quién hará la primera movida del juego.

## Preguntando al Jugador si desea Jugar de Nuevo

```
 42. def jugarDeNuevo():
 43.     # Esta funcion devuelve True (Verdadero) si el jugador desea volver a
jugar, de lo contrario devuelve False (Falso).
 44.     print('¿Deseas volver a jugar? (sí/no)?')
 45.     return input().lower().startswith('s')
```

La función `jugarDeNuevo()` pregunta al jugador si desea jugar de nuevo. Esta función devuelve True si el jugador escribe `'sí'` or `'SÍ'` or `'s'` o cualquier cosa que comience con la letra S. Con cualquier otra respuesta, la función devuelve False. Esta función es igual a la utilizada en el juego del Ahorcado.

## Colocando una Marca en el Tablero

```
 47. def hacerJugada(tablero, letra, jugada):
 48.     tablero[jugada] = letra
```

La función `hacerJugada()` es simple y consiste en sólo una línea. Los parámetros son una lista con diez cadenas llamada tablero, la letra de uno de los jugadores (`'X'` u `'O'`) llamada `letra`, y un espacio en el tablero donde ese jugador quiere jugar (el cual es un entero de `1` a `9`) llamado `jugada`.

Pero espera un segundo. Este código parece cambiar uno de los elementos de la lista `tablero` por el valor en `letra`. Pero como este código pertenece a una función, el parámetro `tablero` será olvidado al salir de esta función y abandonar el entorno de la función. El cambio a `tablero` también será olvidado.

En realidad, esto no es lo que ocurre. Esto se debe a que las listas se comportan en forma especial cuando las pasas como argumentos a funciones. En realidad estás pasando una referencia a la lista y no la propia lista. Vamos a aprender ahora sobre la diferencia entre las listas y las referencias a listas.

## Referencias

Prueba ingresar lo siguiente en la consola interactiva:

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

Esto tiene sentido a partir de lo que sabes hasta ahora. Asignas `42` a la variable `spam`, y luego copias el valor en `spam` y lo asignas a la variable `cheese`. Cuando luego cambias `spam` a `100`, esto no afecta al valor en `cheese`. Esto es porque `spam` y `cheese` son variables diferentes que almacenan valores diferentes

Pero las listas no funcionan así. Cuando asignas una lista a una variable usando el signo =, en realidad asignas a la variable una referencia a esa lista. Una **referencia** es un valor que apunta a un dato. Aquí hay un ejemplo de código que hará que esto sea más fácil de entender. Escribe esto en la consola interactiva:

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

Esto se ve raro. El código sólo modificó la lista `cheese`, pero parece que tanto la lista `cheese` como la lista `spam` han cambiado. Esto se debe a que la variable `spam` no contiene a la propia lista sino una referencia a la misma, como se muestra en la Figura 10-5. La lista en sí misma no está contenida en ninguna variable, sino que existe por fuera de ellas.

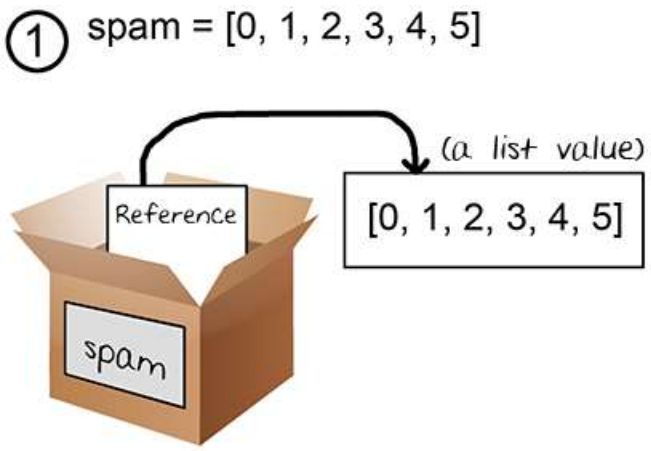



Figura 10-5: Las variables no guardan listas, sino referencias a listas.

Observa que `cheese = spam` copia *la referencia a lista* en `spam` a `cheese`, en lugar de copiar el propio valor de lista. Tanto `spam` como `cheese` guardan una referencia que apunta al mismo valor de lista. Pero sólo hay una lista. No se ha copiado la lista, sino una referencia a la misma. La Figura 10-6 ilustra esta copia.

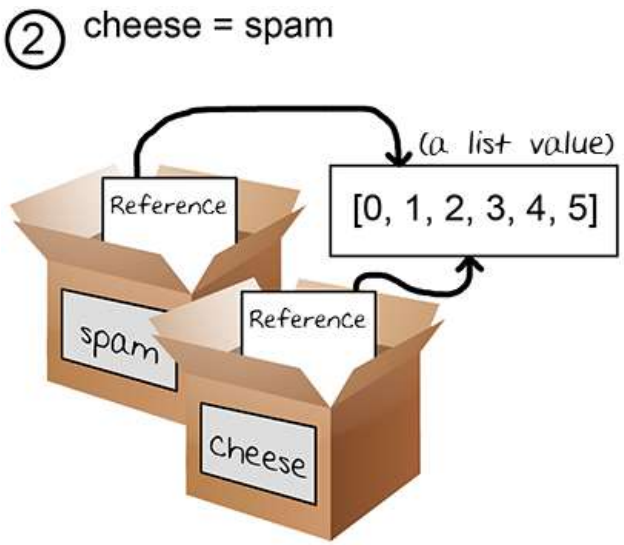



Figura 10-6: Dos variables guardan dos referencias a la misma lista.

Entonces la línea `cheese[1] = '¡Hola!'` cambia la misma lista a la que se refiere `spam`. Es por esto que `spam` parece tener el mismo valor de lista que `cheese`. Ambas tienen referencias que apuntan a la misma lista, como se ve en la Figura 10-7.

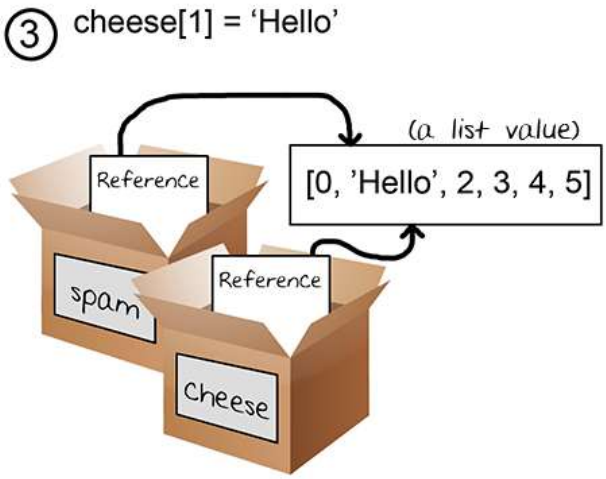



Figura 10-7: Cambio de la lista cambia todas las variables con referencias a la lista.

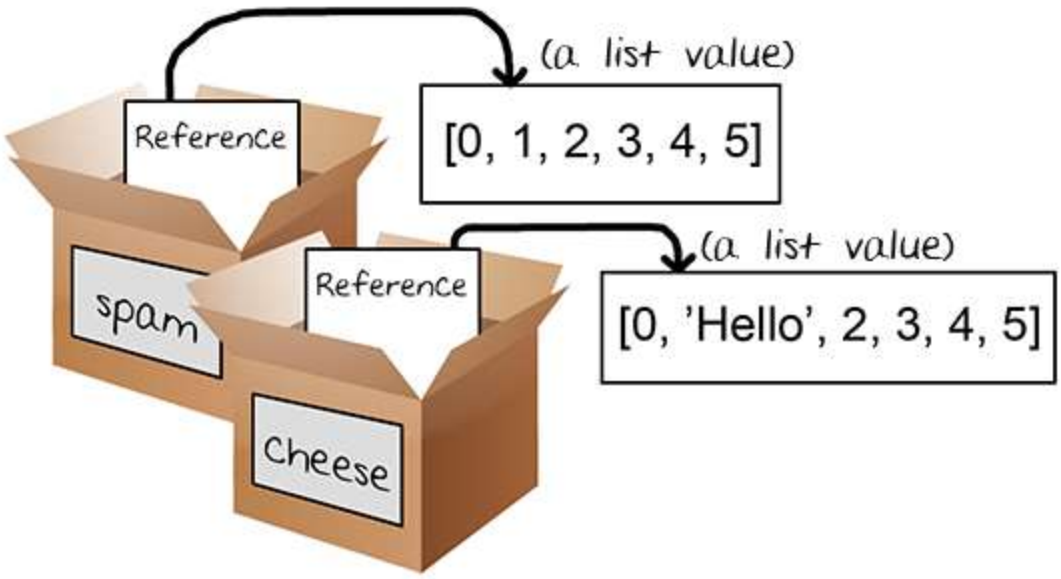
Si quieres que `spam` y `cheese` guarden dos listas diferentes, tienes que crear dos listas diferentes en lugar de copiar una referencia:

```
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = [0, 1, 2, 3, 4, 5]
```

En el ejemplo anterior, spam y cheese almacenan dos listas diferentes (aunque el contenido de ambas sea idéntico). Pero si modificas una de las listas, esto no afectará a la otra porque las variables spam y cheese tienen referencias a dos listas diferentes:

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

La Figura 10-8 muestra como las dos referencias apuntan a dos listas diferentes..




igura 10-8: Dos variables con referencias a dos listas diferentes.

Los diccionarios funcionan de la misma forma. Las variables no almacenan diccionarios, sino que almacenan referencias a diccionarios.

## Usando Referencias a Listas en `hacerJugada()`

Volvamos a la función `hacerJugada()`:

```
47. def hacerJugada(tablero, letra, jugada):
48.     tablero[jugada] = letra
```

Cuando un valor de lista se pasa por el parámetro `tablero`, la variable local de la función es en realidad una copia de la referencia a la lista, no una copia de la lista. Pero una copia de la referencia sigue apuntando a la misma lista a la que apunta la referencia original. Entonces cualquier cambio a `tablero` en esta función ocurrirá también en la lista original. Así es cómo la función `hacerJugada()` modifica la lista original.

Los parámetros `letra` y `jugada` son copias de los valores cadena y entero que pasamos. Como son copias de valores, si modificamos `letra` o jugada en esta función, las variables originales que usamos al llamar a `hacerJugada()` no registrarán cambios.

## Comprobando si el Jugador Ha Ganado

```
50. def esGanador(ta, le):
```

```
 51.     # Dado un tablero y la letra de un jugador, devuelve True (verdadero)
si el mismo ha ganado.
 52.     # Utilizamos reemplazamos tablero por ta y letra por le para no
escribir tanto.
 53.     return ((ta[7] == le and ta[8] == le and ta[9] == le) or # horizontal
superior
 54.     (ta[4] == le and ta[5] == le and ta[6] == le) or # horizontal medio
 55.     (ta[1] == le and ta[2] == le and ta[3] == le) or # horizontal inferior
 56.     (ta[7] == le and ta[4] == le and ta[1] == le) or # vertical izquierda
 57.     (ta[8] == le and ta[5] == le and ta[2] == le) or # vertical medio
 58.     (ta[9] == le and ta[6] == le and ta[3] == le) or # vertical derecha
 59.     (ta[7] == le and ta[5] == le and ta[3] == le) or # diagonal
 60.     (ta[9] == le and ta[5] == le and ta[1] == le)) # diagonal
```

Las líneas 53 a 60 en la función `esGanador()` son el realidad una larga sentencia `return`. Los nombres `ta` y `le` son abreviaturas de los parámetros `tablero` y `letra` para no tener que escribir tanto en esta función. Recuerda, a Python no le importa qué nombres uses para tus variables.

Hay ocho posibles formas de ganar al Ta Te Ti. Puedes formar una línea horizontal arriba, al medio o abajo. O puedes formar una línea vertical a la izquierda, al medio o a la derecha. O puedes formar cualquiera de las dos diagonales.

Fíjate que cada línea de la condición comprueba si los tres espacios son iguales a la letra pasada (combinados con el operador and) y usamos el operador or para combinar las ocho diferentes formas de ganar. Esto significa que sólo una de las ocho formas necesita ser verdadera para que podamos afirmar que el jugador a quien pertenece la letra en `le` es el ganador.

Supongamos que `le` es `'O'`, y el `tablero` se ve así:

```
   |   |
 X |   |
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
 O | O | O
   |   |
```

Así es como se evaluaría la expresión luego de la palabra reservada `return` en la línea 53:

```
53.     return ((bo[7] == le and bo[8] == le and bo[9] == le) or # across the
top
```

```
54.     (ta[4] == le and ta[5] == le and ta[6] == le) or # across the middle
55.     (ta[1] == le and ta[2] == le and ta[3] == le) or # across the bottom
56.     (ta[7] == le and ta[4] == le and ta[1] == le) or # down the left side
57.     (ta[8] == le and ta[5] == le and ta[2] == le) or # down the middle
58.     (ta[9] == le and ta[6] == le and ta[3] == le) or # down the right side
59.     (ta[7] == le and ta[5] == le and ta[3] == le) or # diagonal
60.     (ta[9] == le and ta[5] == le and ta[1] == le)) # diagonal
```

Primero Python reemplazará las variables `ta` y `le` por los valores que contienen::

```
    return (('X' == 'O' and ' ' == 'O' and ' ' == 'O') or
    (' ' == 'O' and 'X' == 'O' and ' ' == 'O') or
    ('O' == 'O' and 'O' == 'O' and 'O' == 'O') or
    ('X' == 'O' and ' ' == 'O' and 'O' == 'O') or
    (' ' == 'O' and 'X' == 'O' and 'O' == 'O') or
    (' ' == 'O' and ' ' == 'O' and 'O' == 'O') or
    ('X' == 'O' and 'X' == 'O' and 'O' == 'O') or
    (' ' == 'O' and 'X' == 'O' and 'O' == 'O'))
```

A continuación, Python evaluará todas las comparaciones == dentro de los paréntesis a un valor Booleano:

```
    return ((False and False and False) or
    (False and False and False) or
    (True and True and True) or
    (False and False and True) or
    (False and False and True) or
    (False and False and True) or
    (False and False and True) or
    (False and False and True))
```

Luego el intérprete Python evaluará todas estas expresiones dentro de los paréntesis:

```
    return ((False) or
    (False) or
    (True) or
    (False) or
    (False) or
    (False) or
    (False) or
    (False))
```

Como ahora hay sólo un valor dentro del paréntesis, podemos eliminarlos:

```
    return (False or
    False or
    True or
    False or
    False or
    False or
    False or
    False)
```

Ahora evaluamos la expresión conectada por todos los operadores `or`:

```
    return (True)
```

Una vez más, eliminamos los paréntesis y nos quedamos con un solo valor:

```
    return True
```

Entonces dados estos valores para ta y le, la expresión se evaluaría a `True`. Así es cómo el programa puede decir si uno de los jugadores ha ganado el juego.

## *Duplicando los Datos del Tablero*

```
62. def obtenerDuplicadoTablero(tablero):
63.     # Duplica la lista del tablero y devuelve el duplicado.
64.     dupTablero = []
65.
66.     for i in tablero:
67.         dupTablero.append(i)
68.
69.     return dupTablero
```

La función `obtenerDuplicadoTablero()` está aquí para que podamos fácilmente hacer una copia de una dada lista de 10 cadenas que representa un tablero de Ta Te Ti en nuestro juego. Algunas veces querremos que nuestro algoritmo IA haga modificaciones temporarias a una copia provisoria del tablero sin cambiar el tablero original. En ese caso, llamaremos a esta función para hacer una copia de la lista del tablero. La nueva lista se crea en la línea 64, con los corchetes `[]` de lista vacía.

Pero la lista almacenada en `dupTablero` en la línea 64 es sólo una lista vacía. El bucle `for` recorre el parámetro `tablero`, agregando una copia de los valores de cadena en el tablero original al tablero duplicado. Finalmente, después del bucle, se devuelve `dupTablero`. La función `obtenerDuplicadoTablero()` construye una copia del tablero original y devuelve una referencia a este nuevo tablero, y no al original.

## Comprobando si un Espacio en el Tablero está Libre

```
 71. def hayEspacioLibre(tablero, jugada):
 72.     # Devuelte true si hay espacio para efectuar la jugada en el tablero.
 73.     return tablero[jugada] == ' '
```

Esta es una función simple que, dado un tablero de Ta Te Ti y una posible jugada, confirmará si esa jugada está disponible o no. Recuerda que los espacios libres en la lista tablero se indican como una cadena con un espacio simple. Si el elemento en el índice del espacio indicado no es igual a una cadena con un espacio simple, el espacio está ocupado y no es una jugada válida.

## Permitiendo al Jugador Ingresar Su Jugada

```
 75. def obtenerJugadaJugador(tablero):
 76.     # Permite al jugador escribir su jugada.
 77.     jugada = ' '
 78.     while jugada not in '1 2 3 4 5 6 7 8 9'.split() or not
hayEspacioLibre(tablero, int(jugada)):
 79.         print('¿Cuál es tu próxima jugada? (1-9)')
 80.         jugada = input()
 81.     return int(jugada)
```

La función `obtenerJugadaJugador()` pide al jugador que ingrese el número del espacio en el que desea jugar. El bucle se asegura de que la ejecución no prosiga hasta que el jugador haya ingresado un entero de 1 a 9. También comprueba que el espacio no esté ocupado, dado el tablero de Ta Te Ti pasado a la función en el parámetro tablero.

Las dos líneas de código dentro del bucle `while` simplemente piden al jugador que ingrese un número de 1 a 9. La condición de la línea 78 es True si cualquiera de las expresiones *a la izquierda o a la derecha* del operador or es `True`.

La expresión en el lado *izquierdo* comprueba si la jugada ingresada por el jugador es igual a `'1'`, `'2'`, `'3'`, y así hasta `'9'` mediante la creación de una lista con estas cadenas (usando el método `split()`) y comprobando si la `jugada` está en esta lista.

`'1 2 3 4 5 6 7 8 9'.split()` se evalúa a la lista `['1', '2', '3', '4', '5', '6', '7', '8', '9']`, pero es más fácil de escribir.

La expresión sobre el lado *derecho* comprueba si la jugada que el jugador ingresó es un espacio libre en el tablero. Lo comprueba llamando a la función `hayEspacioLibre()`. Recuerda que `hayEspacioLibre()` devolverá `True` si la jugada que le hemos pasado está disponible en el

tablero. Nota que `hayEspacioLibre()` espera un entero en el parámetro jugada, así que empleamos la función `int()` para evaluar la forma entera de `jugada`.

Los operadores `not` se agregan a ambos lados de modo que la condición será `True` cuando cualquiera de estos requerimientos deje de cumplirse. Esto hará que el bucle pida al jugador una nueva jugada una y otra vez hasta que la jugada ingresada sea válida.

Finalmente, en la línea 81, se devuelve la forma entera de la jugada ingresada por el jugador. Recuerda que `input()` devuelve una cadena, así que la función `int()` es llamada para devolver la forma entera de la cadena.

## Evaluación en Cortocircuito

Puede ser que hayas notado un posible problema problema en nuestra función `obtenerJugadaJugador()`. ¿Qué pasaría si el jugador ingresara `'Z'` o alguna otra cadena no entera? La expresión `jugada not in '1 2 3 4 5 6 7 8 9'.split()` sobre el lado izquierdo devolvería `False` de acuerdo con lo esperado, y entonces evaluaríamos la expresión sobre el lado derecho del operador `or`.

You may have noticed there's a possible problem in the `getPlayerMove()` function. What if the player typed in `'Z'` or some other non-integer string? The expression `move not in '1 2 3 4 5 6 7 8 9'.split()` on the left side of `or` would return `False` as expected, and then Python would evaluate the expression on the right side of the `or` operator.

Pero llamar a `int('Z')` ocasionaría un error. Python muestra este error porque la función `int()` sólo puede tomar cadenas o caracteres numéricos, tales como `'9'` o `'0'`, no cadenas como 'Z'.

Como un ejemplo de este tipo de error, prueba ingresar esto en la consola interactiva:

```
>>> int('42')
42
>>> int('Z')
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    int('Z')
ValueError: invalid literal for int() with base 10: 'Z'
```

Pero cuando juegas al Ta Te Ti e intentas ingresar `'Z'` en tu jugada, este error no ocurre. La razón de esto es que la condición del bucle `while` está siendo cortocircuitada.

**Evaluar en cortocircuito** quiere decur que como el lado izquierdo de la palabra reservada `or` (`jugada not in '1 2 3 4 5 6 7 8 9'.split()`) se evalúa a `True`, el intérprete Python saba que la expresión completa será evaluada a `True`. No importa si la expresión sobre el lado derecho

de la palabra reservada or se evalúa a `True` o `False`, porque sólo uno de los valores junto al operador or precisa ser `True`.

Piensa en esto: La expresión `True or False` se evalúa a `True` y la expresión `True or True` también se evalúa a `True`. Si el valor sobre el lado izquierdo es `True`, no importa qué valor esté sobre el lado derecho:

`False and <<<anything>>>` always evaluates to `False`

`True or <<<anything>>>` always evaluates to `True`

Entonces Python no comprueba el resto de la expresión y ni siquiera se molesta en evaluar la parte `not hayEspacioLibre(tablero, int(jugada))`. Esto significa las funciones `int()` y `hayEspacioLibre()` nunca son llamadas mientras `jugada not in '1 2 3 4 5 6 7 8 9'.split()` sea `True`.

Esto funciona bien para el programa, pues si la expresión del lado derecho es `True` entonces jugada no es una cadena en forma de número. Esto hace que int() devuelva un error. Las únicas veces que `jugada not in '1 2 3 4 5 6 7 8 9'.split()` se evalúa a `False` son cuando jugada no es una cadena compuesta por un único dígito. En ese caso, la llamada a `int()` no nos daría un error.

## *Un Ejemplo de Evaluación en Cortocircuito*

Aquí hay un pequeño programa que sirve como un buen ejemplo de evaluación en cortocircuito. Prueba escribir lo siguiente en la consola interactiva:

```
>>> def DevuelveTrue():
       print('DevuelveTrue() ha sido llamada.')
       return True

>>> def DevuelveFalse():
       print('DevuelveFalse() ha sido llamada.')
       return False

>>> DevuelveTrue()
DevuelveTrue() ha sido llamada.
True
>>> DevuelveFalse()
DevuelveFalse() ha sido llamada.
False
```

Cuando `DevuelveTrue()` es llamada, IDLE imprime `'DevuelveTrue() ha sido llamada.'` y también muestra el valor retornado por `DevuelveTrue()`. Lo mismo ocurre con `DevuelveFalse()`.

Ahora prueba escribir lo siguiente en la consola interactiva.

```
>>> DevuelveFalse() or DevuelveTrue()
DevuelveFalse() ha sido llamada.
DevuelveTrue() ha sido llamada.
True
>>> DevuelveTrue() or DevuelveFalse()
DevuelveTrue() ha sido llamada.
True
```

La primera parte parece razonable: La expresión `DevuelveFalse() or DevuelveTrue()` llama a ambas funciones, por lo que puedes ver ambos mensajes impresos.

Pero la segunda expresión sólo muestra `'DevuelveTrue() ha sido llamada.'` y no `'DevuelveFalse() ha sido llamada.'`. Esto se debe a que Python no ha llamado a `DevuelveFalse()`. Como el lado izquierdo del operador `or` es `True`, el resultado de `DevuelveFalse()` es irrelevante por lo que Python no se molesta en llamarla. La evaluación ha sido cortocircuitada.

Lo mismo ocurre con el operador `and`. Prueba escribir lo siguiente en la consola interactiva:

```
>>> ReturnsTrue() and ReturnsTrue()
ReturnsTrue() was called.
ReturnsTrue() was called.
True
>>> ReturnsFalse() and ReturnsFalse()
ReturnsFalse() was called.
False
```

Si el lado izquierdo del operador `and` es `False`, entonces la expresión completa será `False`. No importa lo que sea el lado derecho del operador `and`, de modo que Python no se molesta en evaluarlo. Tanto `False and True` como `False and False` se evalúan a `False`, por lo que Python cortocircuita la evaluación.

## *Eligiendo una Jugada de una Lista de Jugadas*

```
83. def elegirAzarDeLista(tablero, listaJugada):
84.     # Devuelve una jugada válida en el tablero de la lista recibida.
85.     # Devuelve None si no hay ninguna jugada válida.
86.     jugadasPosibles = []
87.     for i in listaJugada:
88.         if hayEspacioLibre(tablero, i):
89.             jugadasPosibles.append(i)
```

La función `elegirAzarDeLista()` es útil para el código IA más adelante en el programa. El parámetro tablero es una lista de cadenas que representa un tablero de Ta Te Ti. El segundo parámetro `listaJugada` es una lista de enteros con posibles espacios entre los cuales se puede elegir. Por ejemplo, si `listaJugada` is `[1, 3, 7, 9]`, eso significa que `elegirAzarDeLista()` debería devolver el entero correspondiente a una de las esquinas.

Sin embargo, `elegirAzarDeLista()` comprobará primero que es válido realizar una jugada en ese espacio. La lista `jugadasPosibles` comienza siendo una lista vacía. El bucle `for` itera sobre `listaJugada`. Las jugadas para las cuales `hayEspacioLibre()` devuelve `True` se agregan a `jugadasPosibles` usando el método `append()`.

```
    if len(jugadasPosibles) != 0:
        return random.choice(jugadasPosibles)
    else:
        return None
```

En este punto, la lista `jugadasPosibles` contiene todas las jugadas que estaban en `listaJugada` y también son espacios libres en la lista tablero. Si la lista no está vacía, hay al menos una jugada posible.

La lista podría estar vacía. Por ejemplo, si `listaJugada` fuera `[1, 3, 7, 9]` pero todas las esquinas del `tablero` estuviesen tomadas, la lista `jugadasPosibles` sería `[]`. En ese caso, `len(jugadasPosibles)` se evaluaría a `0` y la función devolvería el valor `None`. La próxima sección explica el valor `None`.

## El Valor None

El **valor None** representa la ausencia de un valor. `None` es el único valor del tipo de datos `NoneType`. Puede ser útil emplear el valor `None` cuando necesites un valor que exprese “no existe” o “ninguno de los anteriores”.

Pongamos por caso que tienes una variable llamada `respuestaExámen` para guardar la respuesta a una pregunta de selección múltiple. La variable podría contener `True` o `False` para indicar la respuesta del usuario. Podrías asignar `None` a `respuestaExámen` si el usuario saltease la pregunta sin responderla. Usar `None` es una forma clara de indicar que el usuario no ha respondido la pregunta.

Las funciones que retornan llegando al final de la función (es decir, sin alcanzar una sentencia return) devolverán `None`. El valor `None` se escribe sin comillas, con una “N” mayúscula y las letras “one” en minúsculas.

Como una nota al margen, `None` no se muestra en la consola interactiva como ocurriría con otros valores:

```
>>> 2 + 2
4
>>> 'Esto es una cadena.'
'Esto es una cadena.'
>>> None
```

Las funciones que aparentan no devolver nada en realidad devuelven el valor `None`. Por ejemplo, `print()` devuelve `None`:

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

## *Creando la Inteligencia Artificial de la Computadora*

```
 96. def obtenerJugadaComputadora(tablero, letraComputadora):
 97.     # Dado un tablero y la letra de la computadora, determina que jugada
efectuar.
 98.     if letraComputadora == 'X':
 99.         letraJugador = 'O'
100.     else:
101.         letraJugador = 'X'
```

La función `obtenerJugadaComputadora()` contiene al código de la IA. El primer argumento es un tablero de Ta Te Ti en el parámetro `tablero`. El segundo es la letra correspondiente a la computadora, sea `'X'` u `'O'` en el parámetro `letraComputadora`. Las primeras líneas simplemente asignan la otra letra a una variable llamada `letraJugador`. De esta forma el mismo código puede usarse independientemente de si la computadora es X u O.

La función devuelve un entero de 1 a 9 que representa el espacio en el que la computadora hará su jugada.

Recuerda cómo funciona el algoritmo del Ta Te Ti:

- Primero, ver si hay una jugada con que la computadora pueda ganar el juego. Si la hay, hacer esa jugada. En caso contrario, continuar al segundo paso.
- Segundo, ver si hay una jugada con la que el jugador pueda vencer a la computadora. Si la hay, la computadora debería jugar en ese lugar para bloquear al jugador. En caso contrario, continuar al tercer paso.

- Tercero, comprobar si alguna de las esquinas (espacios 1, 3, 7, o 9) está disponible. Si ninguna esquina está disponible, continuar al cuarto paso.
- Cuarto, comprobar si el centro está libre. Si lo está, jugar allí. En caso contrario, continuar al quinto paso.
- Quinto, jugar sobre cualquiera de los lados (espacios 2, 4, 6 u 8). No hay más pasos, pues si hemos llegado al quinto paso significa que sólo quedan los espacios sobre los lados.

## *La Computadora Comprueba si puede Ganar en Una Jugada*

```
103.     # Aquí está nuestro algoritmo para nuestra IA (Inteligencia Artifical)
del Ta Te Ti.
104.     # Primero, verifica si podemos ganar en la próxima jugada
105.     for i in range(1, 10):
106.         copia = obtenerDuplicadoTablero(tablero)
107.         if hayEspacioLibre(copia, i):
108.             hacerJugada(copia, letraComputadora, i)
109.             if esGanador(copia, letraComputadora):
110.                 return i
```

Antes que nada, si la computadora puede ganar en la siguiente jugada, debería hacer la jugada ganadora inmediatamente. El bucle `for` que empieza en la línea 105 itera sobre cada posible jugada de 1 a 9. El código dentro del bucle simula lo que ocurriría si la computadora hiciera esa jugada.

La primera línea en el bucle (línea 106) crea una copia de la lista `tablero`. Esto es para que la jugada simulada dentro del bucle no modifique el tablero real de Ta Te Ti guardado en la variable `tablero`. La función `obtenerDuplicadoTablero()` devuelve una copia idéntica pero independiente del tablero.

La línea 107 comprueba si el espacio está libre y, si es así, simula hacer la jugada en la copia del tablero. Si esta jugada resulta en una victoria para la computadora, la la función devuelve el entero correspondiente a esa jugada.

Si ninguna de las jugadas posibles resulta en una victoria, el bucle concluye y la ejecución del programa continúa en la línea 113.

## *La Computadora Comprueba si el Jugador puede Ganar en Una Jugada*

```
112.     # Verifica si el jugador podría ganar en su próxima jugada, y lo
bloquea.
113.     for i in range(1, 10):
```

```
114.         copia = obtenerDuplicadoTablero(tablero)
115.         if hayEspacioLibre(copia, i):
116.             hacerJugada(copia, letraJugador, i)
117.             if esGanador(copia, letraJugador):
118.                 return i
```

A continuación, el código simula un movimiento del jugador en cada uno de los espacios. Este código es similar al bucle anterior, excepto que es la letra del jugador que se coloca sobre la copia del tablero. Si la función `esGanador()` muestra que el jugador ganaría con este movimiento, la computadora devuelve esta jugada para bloquear la victoria del jugador.

Si el jugador humano no puede ganar en la siguiente movida, el bucle `for` eventualmente concluye y la ejecución del programa continúa en la línea 121.

## *Comprobando las Esquinas, Centro y Espacios sobre los Lados (en ese Orden)*

```
120.     # Intenta ocupar una de las esquinas de estar libre.
121.     jugada = elegirAzarDeLista(tablero, [1, 3, 7, 9])
122.     if jugada != None:
123.         return jugada
```

La llamada a `elegirAzarDeLista()` con la lista `[1, 3, 7, 9]` asegura que la función devuelva el entero de una de de las esquinas: `1`, `3`, `7`, ó `9`. Si todas las esquinas están tomadas, la función `elegirAzarDeLista()` devuelve `None` y la ejecución continúa en la línea 126.

```
125.     # De estar libre, intenta ocupar el centro.
126.     if hayEspacioLibre(tablero, 5):
127.         return 5
```

Si ninguna de las esquinas está disponible, la línea 127 intentará jugar en el centro. Si el centro no está libre, la ejecución continúa sobre la línea 130.

```
129.     # Ocupa alguno de los lados.
130.     return elegirAzarDeLista(tablero, [2, 4, 6, 8])
```

Este código también llama a `elegirAzarDeLista()`, sólo que le pasamos una lista con los espacios sobre los lados (`[2, 4, 6, 8]`). Esta función no devolverá `None` pues los espacios sobre los lados son los únicos que pueden estar disponibles. Esto concluye la función `obtenerJugadaComputadora()` y el algoritmo IA.

## Comprobando si el Tablero está Lleno

```
132. def tableroCompleto(tablero):
133.     # Devuelve True si cada espacio del tablero fue ocupado, caso
contrario devuele False.
134.     for i in range(1, 10):
135.         if hayEspacioLibre(tablero, i):
136.             return False
137.     return True
```

La última función es `tableroCompleto()`. Esta función devuelve `True` si la lista tablero pasada como argumento tiene una `'X'` o una `'O'` en cada índice (excepto por el índice `0`, que es ignorado por el código). Si hay al menos un casillero en el tablero con espacio simple `' '` asignado, esta función devolverá `False`.

El bucle for nos permite comprobar los espacios 1 a 9 en el tablero de Ta Te Ti. Apenas encuentra un espacio libre en el tablero (es decir, cuando `hayEspacioLibre(tablero, i)` devuelve `True`) la función `tableroCompleto()` devuelve `False`.

Si la ejecución concluye todas las operaciones del bucle, significa que ninguno de los espacios está libre. Entonces se ejecutará la línea 137 y `devolverá True`.

## El Inicio del Juego

```
140. print('¡Bienvenido al Ta Te Ti!')
```

La línea 140 es la primera línea que no está dentro de una función, de modo que es la primera línea de código que se ejecuta al entrar a este programa. Consiste en el saludo al jugador.

```
142. while True:
143.     # Resetea el tablero
144.     elTablero = [' '] * 10
```

Este bucle while tiene al valor `True` por condición, y continuará iterando hasta que la ejecución llegue a una sentencia `break`. La línea 144 configura el tablero principal de Ta Te Ti que usaremos, al cual llamaremos `elTablero`. Es una lista de 10 cadenas, donde cada una de ellas es un espacio simple `' '`.

En lugar de escribir esta lista completa, la línea 44 usa replicación de listas. Es más corto y claro escribir `[' '] * 10` que escribir `[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']`.

## Decidiendo la Letra del Jugador y Quién Comienza

```
145.     letraJugador, letraComputadora = ingresaLetraJugador()
```

La función `ingresaLetraJugador()` permite al jugador elegir si quiere ser X u O. La función devuelve una lista de dos cadenas, la cual puede ser `['X', 'O']` o `['O', 'X']`. El truco de asignación múltiple asignará `letraJugador` al primer elemento en la lista devuelta y `letraComputadora` al segundo.

```
146.     turno = quienComienza()
147.     print(turno + ' irá primero.')
148.     juegoEnCurso = True
```

La función `quienComienza()` decide aleatoriamente quién comienza, y devuelve la cadena `'El jugador'` o bien `'La computadora'` y la línea 147 comunica al jugador quién comenzará.

## *Ejecutando el Turno del Jugador*

```
150.     while juegoEnCurso:
```

El bucle de la línea 150 continuará alternando entre el código del turno del jugador y el del turno de la computadora, mientras la `juegoEnCurso` tenga asignado el valor `True`.

```
151.         if turno == 'El jugador':
152.             # Turno del jugador
153.             dibujarTablero(elTablero)
154.             jugada = obtenerJugadaJugador(elTablero)
155.             hacerJugada(elTablero, letraJugador, jugada)
```

El valor en la variable `turno` es originalmente asignado por llamada a la función `quienComienza()` en la línea 146. Su valor original es `'El jugador'` o `'La computadora'`. Si turno es igual a `'La computadora'`, la condición es `False` y la ejecución salta a la línea 169.

Primero, la línea 153 llama a la función `dibujarTablero()` pasándole la variable `elTablero` para dibujar el tablero en la pantalla. Entonces la función `obtenerJugadaJugador()` permite al jugador ingresar su jugada (y también comprueba que sea una movida válida). La función `hacerJugada()` actualiza `elTablero` para reflejar esta jugada.

```
157.             if esGanador(elTablero, letraJugador):
158.                 dibujarTablero(elTablero)
159.                 print('¡Felicidades, has ganado!')
160.                 juegoEnCurso = False
```

Luego de que el jugador ha jugado, la computadora debería comprobar si ha ganado el juego. Si la función `esGanador()` devuelve `True`, el código del bloque if muestra el tablero ganador e imprime un mensaje comunicando al jugador que ha ganado.

Se asigna el valor `False` a la variable `juegoEnCurso` de modo que la ejecución no continúe con el turno de la computadora.

```
161.             else:
162.                 if tableroCompleto(elTablero):
163.                     dibujarTablero(elTablero)
164.                     print('¡Es un empate!')
165.                     break
```

Si el jugador no ganó con esta última jugada, tal vez esta movida ha llenado el tablero y ocasionado un empate. En este bloque else, la función `tableroCompleto()` devuelve `True` si no hay más movimientos disponibles. En ese caso, el bloque `if` que comienza en la línea 162 muestra el tablero empatado y comunica al jugador que ha habido un empate. La ejecución sale entonces del bucle `while` y salta a la línea 186.

```
166.                 else:
167.                     turno = 'La computadora'
```

Si el jugador no ha ganado u ocasionado un empate, la línea 167 asigna `'La computadora'` a la variable turno, de modo que ejecute el código para el turno de la computadora en la siguiente iteración.

## *Ejecutando el Turno de la Computadora*

Si la variable `turno` no es `'El Jugador'` para la condición en la línea 151, entonces es el turno de la computadora. El código en este bloque else es similar al código para el turno del jugador.

```
169.         else:
170.             # Turno de la computadora
171.             jugada = obtenerJugadaComputadora(elTablero, letraComputadora)
172.             hacerJugada(elTablero, letraComputadora, jugada)
173.
174.             if esGanador(elTablero, letraComputadora):
175.                 dibujarTablero(elTablero)
176.                 print('¡La computadora te ha vencido! Has perdido.')
177.                 juegoEnCurso = False
178.             else:
179.                 if tableroCompleto(elTablero):
180.                     dibujarTablero(elTablero)
181.                     print('¡Es un empate!')
```

```
182.                     break
183.                 else:
184.                     turno = 'El jugador'
```

Las líneas 170 a 184 son casi idénticas al código del turno del jugador en las líneas 152 a 167. La única diferencia es que se comprueba si ha habido un empate luego del turno de la computadora en lugar de hacerlo luego del turno del jugador.

Si no existe un ganador y no es un empate, la línea 184 cambia el turno al jugador. No hay más líneas de código dentro del bucle `while`, de modo que la ejecución vuelve a la sentencia `while` en la línea 150.

```
186.     if not jugarDeNuevo():
187.         break
```

Las líneas 186 y 187 se encuentran inmediatamente a continuación del bloque `while` que comienza con la sentencia `while` en la línea 150. Se asigna `False` a `juegoEnCurso` cuando el juego ha terminado, por lo que en este punto se pregunta al jugador si desea jugar de nuevo.

Si `jugarDeNuevo()` devuelve `False`, la condición de la sentencia `if` es `True` (porque el operador `not` invierte el valor Booleano) y se ejecuta la sentencia `break`. Esto interrumpe la ejecución del bucle `while` que había comenzado en la línea 142. Como no hay más líneas de código a continuación de ese bloque `while`, el programa termina.

## *Resumen*

Crear un programa que pueda jugar un juego se reduce a considerar cuidadosamente todas las situaciones posibles en las que la IA pueda encontrarse y cómo responder en cada una de esas situaciones. La IA del Ta Te Ti es simple porque no hay muchos movimientos posibles en Ta Te Ti comparado con un juego como el ajedrez o las damas.

Nuestra IA simplemente comprueba si puede ganar en la próxima jugada. Si no es posible, bloquea la movida del jugador cuando está a punto de ganar. En cualquier otro caso la IA simplemente intenta jugar en cualquier esquina disponible, luego el centro y por último los lados. Este es un algoritmo simple y fácil de seguir.

La clave para implementar nuestro algoritmo IA es hacer una copia de los datos del tablero y simular jugadas sobre la copia. De este modo, el código de IA puede hacer esa jugada en el tablero real. Este tipo de simulación es efectivo a la hora de predecir si una jugada es buena o no.

# Capítulo 11
# PANECILLOS

Temas Tratados En Este Capítulo:
- Operadores de Asignación Aumentada, +=, -=, *=, /=
- La Función `random.shuffle()`
- Los Métodos de Lista `sort()` y `join()`
- Interpolación de Cadenas (también llamado Formateo de Cadenas)
- Indicador de Conversión `%s`
- Bucles Anidados

En este capítulo aprenderás algunos nuevos métodos y funciones que vienen con Python. También aprenderás acerca de operadores de asignación aumentada e interpolación de cadenas. Estos conecptos no te permitirán hacer nada que no pudieras hacer antes, pero son buenos atajos que hacen más fácil escribir tu código.

Panecillos es un juego simple que puedes jugar con un amigo. Tu amigo piensa un número aleatorio de 3 cifras diferentes, y tú intentas adivinar qué número es. Luego de cada intento, tu amigo te dará tres tipos de pistas:

- **Panecillos** – Ninguna de las tres cifras del número que has conjeturado está en el número secreto.
- **Pico** – Una de las cifras está en el número secreto, pero no en el lugar correcto.
- **Fermi** – Tu intento tiene una cifra correcta en el lugar correcto.

Puedes recibir más de una pista luego de un intento. Si el número secreto es 456 y tú conjeturas 546, la pista sería "fermi pico pico". El número 6 da "Fermi" y el 5 y 4 dan "pico pico".

## Prueba de Ejecución

```
Estoy pensando en un número de 3 dígitos. Intenta adivinar cuál es.
Aquí hay algunas pistas:
Cuando digo:    Eso significa:
  Pico         Un dígito es correcto pero en la posición incorrecta.
  Fermi        Un dígito es correcto y en la posición correcta.
  Panecillos       Ningún dígito es correcto.
He pensado un número. Tienes 10 intentos para adivinarlo.
Conjetura #1:
```

```
123
Fermi
Conjetura #2:
453
Pico
Conjetura #3:
425
Fermi
Conjetura #4:
326
Panecillos
Conjetura #5:
489
Panecillos
Conjetura #6:
075
Fermi Fermi
Conjetura #7:
015
Fermi Pico
Conjetura #8:
175
¡Lo has adivinado!
¿Deseas volver a jugar? (sí o no)
no
```

## Código Fuente de Panecillos

Si obtienes errores luego de escribir este código, compara el código que has escrito con el código del libto usando la herramienta diff online en http://invpy.com/es/diff/bagels.

panecillos.py

```
 1. import random
 2. def obtenerNumSecreto(digitosNum):
 3.     # Devuelve un numero de largo digotosNUm, compuesto de dígitos únicos
al azar.
 4.     numeros = list(range(10))
 5.     random.shuffle(numeros)
 6.     numSecreto = ''
 7.     for i in range(digitosNum):
 8.         numSecreto += str(numeros[i])
 9.     return numSecreto
10.
11. def obtenerPistas(conjetura, numSecreto):
12.     # Devuelve una palabra con las pistas Panecillos Pico y Fermi en ella.
```

```
    if conjetura == numSecreto:
        return '¡Lo has adivinado!'

    pista = []

    for i in range(len(conjetura)):
        if conjetura[i] == numSecreto[i]:
            pista.append('Fermi')
        elif conjetura[i] in numSecreto:
            pista.append('Pico')
    if len(pista) == 0:
        return 'Panecillos'

    pista.sort()
    return ' '.join(pista)

def esSoloDigitos(num):
    # Devuelve True si el número se compone sólo de dígitos. De lo
contrario False.
    if num == '':
        return False

    for i in num:
        if i not in '0 1 2 3 4 5 6 7 8 9'.split():
            return False

    return True

def jugarDeNuevo():
    # Esta funcion devuelve True si el jugador desea vovler a jugar, de lo
contrario False.
    print('¿Deseas volver a jugar? (sí o no)')
    return input().lower().startswith('s')

digitosNum = 3
MAXADIVINANZAS = 10

print('Estoy pensando en un número de %s dígitos. Intenta adivinar cuál
es.' % (digitosNum))
print('Aquí hay algunas pistas:')
print('Cuando digo:    Eso significa:')
print('  Pico          Un dígito es correcto pero en la posición
incorrecta.')
print('  Fermi         Un dígito es correcto y en la posición correcta.')
print('  Panecillos    Ningún dígito es correcto.')

while True:
```

```
56.     numSecreto = obtenerNumSecreto(digitosNum)
57.     print('He pensado un número. Tienes %s intentos para adivinarlo.' %
(MAXADIVINANZAS))
58.
59.     numIntentos = 1
60.     while numIntentos <= MAXADIVINANZAS:
61.         conjetura = ''
62.         while len(conjetura) != digitosNum or not esSoloDigitos(conjetura):
63.             print('Conjetura #%s: ' % (numIntentos))
64.             conjetura = input()
65.
66.         pista = obtenerPistas(conjetura, numSecreto)
67.         print(pista)
68.         numIntentos += 1
69.
70.         if conjetura == numSecreto:
71.             break
72.         if numIntentos > MAXADIVINANZAS:
73.             print('Te has quedado sin intentos. La respuesta era %s.' %
(numSecreto))
74.
75.     if not jugarDeNuevo():
76.         break
```

## Diseñando el Programa

El diagrama de flujo en la Figura 11-1 describe qué ocurre en este juego, y en qué orden.

### *Cómo Funciona el Código*

```
 1. import random
 2. def obtenerNumSecreto(digitosNum):
 3.     # Devuelve un numero de largo digotosNUm, compuesto de dígitos únicos
al azar.
```

Al comienzo del programa, importamos el módulo `random`. Luego definimos una función llamada `obtenerNumSecreto()`. La función crea un número secreto sin cifras repetidas. En lugar de números secretos de sólo 3 cifras, el parámetro `digitosNum` permite a la función crear un número secreto con cualquier cantidad de cifras. Por ejemplo, puedes crear un número secreto de cuatro o seis cifras pasando `4` ó `6` en `digitosNum`.


Figura 11-1: Diagrama de flujo para el juego Panecillos.

## *Mezclando un Conjunto de Cifras Únicas*

```
4.     numeros = list(range(10))
5.     random.shuffle(numeros)
```

La expresión de la línea 4 `list(range(10))` siempre se evalúa a `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`. Simplemente es más fácil escribir `list(range(10))`. La variable numeros contiene una lista de las diez posibles cifras.

## La función `random.shuffle()`

La función `random.shuffle()` cambia aleatoriamente el orden de los elementos de una lista. Esta función no devuelve un valor, sino que modifica la lista que se le pasa "in situ". Esto es similar al modo en que la función `hacerJugada()` en el capítulo Ta Te Ti modifica la lista que se le pasa, en lugar de devolver una nueva lista con el cambio. Por eso **no** necesitamos escribir `numeros = random.shuffle(numeros)`.

Experimenta con la función `random.shuffle()` ingresando el siguiente código en la consola interactiva:

```
>>> import random
```

```
>>> spam = list(range(10))
>>> print(spam)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> random.shuffle(spam)
>>> print(spam)
[3, 0, 5, 9, 6, 8, 2, 4, 1, 7]

>>> random.shuffle(spam)
>>> print(spam)
[1, 2, 5, 9, 4, 7, 0, 3, 6, 8]

>>> random.shuffle(spam)
>>> print(spam)
[9, 8, 3, 5, 4, 7, 1, 2, 0, 6]
```

La idea es que el número secreto en Panecillos tenga cifras únicas. El juego Panecillos es mucho más divertido si no tienes cifras duplicadas en el número secreto, como en `'244'` o `'333'`. La función `shuffle()` te ayudará a lograr esto.

## *Obteniendo el Número Secreto a partir de las Cifras Mezcladas*

```
6.      numSecreto = ''
7.      for i in range(digitosNum):
8.          numSecreto += str(numeros[i])
9.      return numSecreto
```

El número secreto será una cadena de las primeras `digitosNum` cifras de la lista mezclada de enteros. Por ejemplo, si la lista mezclada en `numeros` fuese `[9, 8, 3, 5, 4, 7, 1, 2, 0, 6]` y `digitosNum` fuese `3`, entonces la cadena devuelta por `obtenerNumSecreto()` será `'983'`.

Para hacer esto, la variable `numSecreto` comienza siendo una cadena vacía. El bucle `for` en la línea 7 itera `digitosNum` veces. En cada iteración del bucle, el entero en el índice `i` es copiado de la lista mezclada, convertido a cadena, y concatenado al final de `numSecreto`.

Por ejemplo, si numeros se refiere a la lista `[9, 8, 3, 5, 4, 7, 1, 2, 0, 6]`, entonces en la primera iteración, `numeros[0]` (es decir, `9`) será pasado a `str()`, que a su vez devolverá `'9'` el cual es concatenado al final de `numSecreto`. En la segunda iteración, lo mismo ocurre con `numeros[1]` (es decir, `8`) y en la tercera iteración lo mismo ocurre con `numeros[2]` (es decir, `3`). El valor final de `numSecreto` que se devuelve es `'983'`.

Observa que `numSecreto` en esta función contiene una cadena, no un entero. Esto puede parecer raro, pero recuerda que no puedes concatenar enteros. La expresión `9 + 8 + 3` se evalúa a `20`, pero lo que tú quieres ahora es `'9' + '8' + '3'`, que se evalúa a `'983'`.

## Operadores de Asignación Aumentada

El operador += en la línea 8 es uno de los **operadores de asignación aumentada**. Normalmente, si quisieras sumar o concatenar un valor a una variable, usarías un código como el siguiente:

```
spam = 42
spam = spam + 10

eggs = 'Hello '
eggs = eggs + 'world!'
```

Los operadores de asignación aumentada son un atajo que te libera de volver a escribir el nombre de la variable. El siguiente código hace lo mismo que el código de más arriba:

```
spam = 42
spam += 10       # Like spam = spam + 10

eggs = 'Hello '
eggs += 'world!' # Like eggs = eggs + 'world!'
```

Existen otros operadores de asignación aumentada. Prueba ingresar lo siguiente en la consola interactiva:

```
>>> spam = 42
>>> spam -= 2
>>> spam
40
>>> spam *= 3
>>> spam
120
>>> spam /= 10
>>> spam
12.0
```

### *Calculando las Pistas a Dar*

```
11. def obtenerPistas(conjetura, numSecreto):
12.     # Devuelve una palabra con las pistas Panecillos Pico y Fermi en ella.
13.     if conjetura == numSecreto:
14.         return '¡Lo has adivinado!'
```

La función `obtenerPistas()` devolverá una sola cadena con las pistas fermi, pico, y panecillos dependiendo de los parámetros `conjetura` y `numSecreto`. El paso más obvio y sencillo es

comprobar si la conjetura coincide con el número secreto. En ese caso, la línea 14 devuelve `'¡Lo has adivinado!'`.

```
16.     pista = []
17.
18.     for i in range(len(conjetura)):
19.         if conjetura[i] == numSecreto[i]:
20.             pista.append('Fermi')
21.         elif conjetura[i] in numSecreto:
22.             pista.append('Pico')
```

Si la conjetura no coincide con el número secreto, el código debe determinar qué pistas dar al jugador. La lista en `pista` comenzará vacía y se le añadirán cadenas `'Fermi'` y `'Pico'` a medida que se necesite.

Hacemos esto recorriendo cada posible índice en `conjetura` y `numSecreto`. Las cadenas en ambas variables serán de la misma longitud, de modo que la línea 18 podría usar tanto `len(conjetura)` como `len(numSecreto)` y funcionar igual. Como el valor de `i` cambia de `0` a `1` a `2`, y así sucesivamente, la línea 19 comprueba si la primera, segunda, tercera, etc. letra de `conjetura` es la misma que el número correspondiente al mismo índice en de `numSecreto`. Si es así, la línea 20 agregará una cadena `'Fermi'` a pista.

De lo contrario, la línea 21 comprobará si la cifra en la posición `i`-ésima de conjetura existe en algún lugar de `numSecreto`. Si es así, ya sabes que la cifra está en algún lugar del número secreto pero no en la misma posición. Entonces la línea 22 añadirá `'Pico'` a `pista`.

```
23.     if len(pista) == 0:
24.         return 'Panecillos'
```

Si la `pista` está vacía luego del bucle, significa que no hay ninguna cifra correcta en la `conjetura`. En ese caso, la línea 24 devuelve la cadena `'Panecillos'` como única pista.

## El Método de Lista `sort()`

```
26.     pista.sort()
```

Las listas tienen un método llamado `sort()` que reordena los elementos de la lista para dejarlos en orden alfabético o numérico. Prueba escribir lo siguiente en la consola interactiva:

```
>>> spam = ['cat', 'dog', 'bat', 'anteater']
>>> spam.sort()
>>> spam
```

```
['anteater', 'bat', 'cat', 'dog']

>>> spam = [9, 8, 3, 5, 4, 7, 1, 2, 0, 6]
>>> spam.sort()
>>> spam
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

El método `sort()` no devuelve una lista ordenada, sino que reordena la lista sobre la cual es llamado "in situ". De esta forma funciona también el método `reverse()`.

Nunca querrás usar esta línea de código: `return spam.sort()` porque esto devolvería el valor `None` (que es lo que devuelve `sort()`). En lugar de esto probablemente quieras una línea separada `spam.sort()` y luego la línea return `spam`.

La razón por la cual quieres ordenar la lista `pista` es para eliminar la información extra basada en el orden de las pistas. Si `pista` fuese `['Pico', 'Fermi', 'Pico']`, eso permitiría al jugador saber que la cifra central de la conjetura está en la posición correcta. Como las otras dos pistas son Pico, el jugador sabría que todo lo que tiene que hacer es intercambiar la primera cifra con la tercera para obtener el número secreto.

Si las pistas están siempre en orden alfabético, el jugador no puede saber a cuál de los números se refiere la pista Fermi. Así queremos que sea el juego.

## El Método de Cadena `join()`

```
27.     return ' '.join(pista)
```

El método de cadena `join()` devuelve una lista de cadenas agrupada en una única cadena. La cadena sobre la cual se llama a este método (en la línea 27 es un espacio simple, `' '`) aparece entre las cadenas de la lista. Por ejemplo, escribe lo siguiente en la consola interactiva:

```
>>> ' '.join(['Mi', 'nombre', 'es', 'Zophie'])
'Mi nombre es Zophie'
>>> ', '.join(['Vida', 'el Universo', 'y Todo'])
'Vida, el Universo, y Todo'
```

Entonces la cadena que se devuelve en la línea 27 corresponde a todas las cadenas en `pista` agrupadas con un espacio simple entre cada una. El método de cadena `join()` es algo así como el opuesto al método de cadena `split()`. Mientras que `split()` devuelve una lista a través de fragmentar una cadena, `join()` devuelve una cadena a través de agrupar una lista.

### *Comprobando si una Cadena Tiene Sólo Números*

```
29. def esSoloDigitos(num):
30.     # Devuelve True si el número se compone sólo de dígitos. De lo
contrario Falso.
31.     if num == '':
32.         return False
```

La función `esSoloDigitos()` ayuda a determinar si el jugador ha ingresado una conjetura válida. La línea 31 comprueba si `num` es una cadena vacía, en cuyo caso devuelve `False`.

```
34.     for i in num:
35.         if i not in '0 1 2 3 4 5 6 7 8 9'.split():
36.             return False
37.
38.     return True
```

El bucle `for` itera sobre cada caracter en la cadena `num`. El valor de `i` tendrá sólo un caracter en cada iteración. Dentro del bloque `for`, el código comprueba si `i` no existe en la lista devuelta por `'0 1 2 3 4 5 6 7 8 9'.split()`. (El valor devuelto por `split()` es equivalente a `['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']` pero es más fácil de escribir.) Si no lo es, sabemos que uno de los caracteres de `num` no es un dígito. En ese caso, la línea 36 devuelve `False`.

Si la ejecución continúa luego del bucle `for`, sabemos que cada caracter en `num` es una cifra. En ese caso, a línea 38 devuelve `True`.

## *Preguntando si el Jugador Quiere Volver a Jugar*

```
40. def jugarDeNuevo():
41.     # Esta funcion devuelve True si el jugador desea vovler a jugar, de lo
contrario Falso.
42.     print('¿Deseas volver a jugar? (sí o no)')
43.     return input().lower().startswith('s')
```

La función `jugarDeNuevo()` es la misma que has usado en el Ahorcado y el Ta Te Ti. La expresión larga en la línea 43 se evalúa a `True` o `False` basándose en la respuesta dada por el jugador.

## *El Comienzo del Juego*

```
45. digitosNum = 3
46. MAXADIVINANZAS = 10
47.
```

```
48. print('Estoy pensando en un número de %s dígitos. Intenta adivinar cuál
es.' % (digitosNum))
49. print('Aquí hay algunas pistas:')
50. print('Cuando digo:    Eso significa:')
51. print('  Pico          Un dígito es correcto pero en la posición
incorrecta.')
52. print('  Fermi         Un dígito es correcto y en la posición correcta.')
53. print('  Panecillos    Ningún dígito es correcto.')
```

Después de haber definido todas las funciones, aquí comienza el programa. En lugar de usar el entero `3` para la cantidad de cifras en el número secreto, usamos la variable constante `digitosNum`. Lo mismo corre para el uso de `MAXADIVINANZAS` en lugar del entero `10` para la cantidad de conjeturas que se permite al jugador. Ahora será fácil cambiar el número de conjeturas o cifras del número secreto. Sólo precisamos cambiar la línea 45 ó 46 y el resto del programa funcionará sin más cambios.

Las llamadas a la función `print()` explicarán al jugador las reglas de juego y lo que significan las pistas Pico, Fermi, y Panecillos. La llamada a `print()` de la línea 48 tiene `% (digitosNum)` agregado al final y `%s` dentro de la cadena. Esto es una técnica conocida como interpolación de cadenas.

## Interpolación de Cadenas

Interpolación de cadenas es una abreviatura del código. Normalmente, si quieres usar los valores de cadena dentro de variables en otra cadena, tienes que usar el operador concatenación `+` :

```
>>> nombre = 'Alicia'
>>> evento = 'fiesta'
>>> dónde = 'la piscina'
>>> día = 'sábado'
>>> hora = '6:00pm'

>>> print('Hola, ' + name + '. ¿Vendrás a la ' + evento + ' en ' + dónde + '
este ' + día + ' a las ' + hora + '?')
Hola, Alicia. ¿Vendrás a la fiesta en la piscina este sábado a las 6:00pm?
```

Como puedes ver, puede ser difícil escribir una línea que concatena varias cadenas. En lugar de esto, puedes usar **interpolación de cadenas**, lo cual te permite utilizar comodines como `%s`. Estos comodines se llaman **especificadores de conversión**. Luego colocas todos los nombres de variables al final. Cada `%s` es reemplazado por una variable al final de la línea. Por ejemplo, el siguiente código hace lo mismo que el anterior:

As you can see, it can be hard to type a line that concatenates several strings. Instead, you can use **string interpolation**, which lets you put placeholders like `%s`. These placeholders are called **conversion specifiers**. Then put all the variable names at the end. Each `%s` is replaced with a variable at the end of the line. For example, the following code does the same thing as the previous code:

```
>>> nombre = 'Alicia'
>>> evento = 'fiesta'
>>> dónde = 'la piscina'
>>> día = 'sábado'
>>> hora = '6:00pm'

>>> print(Hola, %s. ¿Vendrás a la %s en %s este %s a las %s?' % (name, event,
where, day, time))
Hola, Alicia. ¿Vendrás a la fiesta en la piscina este sábado a las 6:00pm?
```

La interpolación de cadenas puede hacer tu código mucho más fácil de escribir. El primer nombre de variable corresponde al primer `%s`, la segunda variable va con el segundo `%s` y así sucesivamente. debes tener tantos especificadores de conversión %s como variables.

Otro beneficio de usar interpolación de cadenas en lugar de concatenación es que la interpolación funciona con cualquier tipo de datos, no sólo cadenas. Todos los valores se convierten automáticamente al tipo de datos cadena. Si concatenases un entero a una cadena, obtendrías este error:

```
>>> spam = 42
>>> print('Spam == ' + spam)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

La concatenación de cadenas sólo funciona para dos o más cadenas, pero `spam` es un entero. Tendrías que recordar escribir `str(spam)` en lugar de `spam`. Pero con interpolación de cadenas, esta conversión a cadenas se realiza automáticamente. Prueba escribir lo siguiente en la consola interactiva:

```
>>> spam = 42
>>> print('Spam es %s' % (spam))
Spam is 42
```

La interpolación de cadenas también se conoce como **formateo de cadenas**.

## *Creando el Número Secreto*

```
55. while True:
56.     numSecreto = obtenerNumSecreto(digitosNum)
57.     print('He pensado un número. Tienes %s intentos para adivinarlo.' %
(MAXADIVINANZAS))
58.
59.     numIntentos = 1
60.     while numIntentos <= MAXADIVINANZAS:
```

La línea 55 es un bucle `while` infinito que tiene una condición `True`, por lo que seguirá repitiéndose eternamente hasta que se ejecute una sentencia break. Dentro de este bucle infinito, se obtiene un número secreto de la función `obtenerNumSecreto()`, pasándole a la misma `digitosNum` para indicar cuántas cifras debe tener el número. Este número secreto es asignado a `numSecreto`. Recuerda, el valor en `numSecreto` es una cadena, no un entero.

La línea 57 indica al jugador cuántas cifras hay en el número secreto usando interpolación de cadena en lugar de concatenación. La línea 59 asigna `1` a la variable `numIntentos` para indicar que este es el primer intento. Entonces la línea 60 tiene un nuevo bucle `while` que se ejecuta mientras `numIntentos` sea menor o igual que `MAXADIVINANZAS`.

## Obteniendo la Conjetura del Jugador

```
61.         conjetura = ''
62.         while len(conjetura) != digitosNum or not esSoloDigitos(conjetura):
63.             print('Conjetura #%s: ' % (numIntentos))
64.             conjetura = input()
```

La variable `conjetura` almacenará la conjetura del jugador devuelta por `input()`. El código continúa iterando y pidiendo al jugador una nueva conjetura hasta que el jugador ingrese una conjetura válida. Una conjetura válida está compuesta únicamente por cifras y la misma cantidad de cifras que el número secreto. Esta es la función que cumple el bucle `while` que comienza en la línea 62.

Se asigna una cadena vacía a la variable `conjetura` en la línea 61, de modo que la condición del bucle `while` sea `False` en la primera comprobación, asegurando que la ejecución entre al bucle.

## Obteniendo las Pistas para la Conjetura del Jugador

```
66.         pista = obtenerPistas(conjetura, numSecreto)
67.         print(pista)
68.         numIntentos += 1
```

Una vez que la ejecución pasa el bucle `while` que comienza la línea 62, la variable `conjetura` contiene un número válido. El mismo se pasa junto con `numSecreto` a la función `obtenerPistas()`. Esta función devuelve una cadena de pistas, las cuales se muestran al jugador en la línea 67. La línea 68 incrementa `numIntentos` usando el operador de asignación aumentada para la suma.

## Comprobando si el Jugador ha Ganado o Perdido

Observa que este segundo bucle `while` sobre la línea 60 se encuentra dentro de otro bucle `while` que comienza más arriba en la línea 55. Estos bucles dentro de bucles se llaman **bucles anidados**. Cualquier sentencia break o continue sólo tendrá efecto sobre el bucle interno, y no afectará a ninguno de los bucles externos.

```
70.         if conjetura == numSecreto:
71.             break
72.         if numIntentos > MAXADIVINANZAS:
73.             print('Te has quedado sin intentos. La respuesta era %s.' %
(numSecreto))
```

Si `conjetura` es igual a `numSecreto`, el jugador ha adivinado correctamente el número secreto y la línea 71 sale del bucle `while` que había comenzado en la línea 60.

Si no lo es, la ejecución continúa a la línea 72, donde comprueba si al jugador se le han acabado los intentos. Si es así, el programa avisa al jugador que ha perdido.

En este punto, la ejecución retorna al bucle `while` en la línea 60 donde permite al jugador tomar otro intento. Si el jugador se queda sin intentos (o si ha salido del bucle con la sentencia `break` de la línea 71), la ejecución continuará más allá del bucle y hasta la línea 75.

## Preguntando al Jugador si desea Volver a Jugar

```
75.     if not jugarDeNuevo():
76.         break
```

La línea 75 pregunta al jugador si desea jugar otra vez llamando a la función `jugarDeNuevo()`. Si `jugarDeNuevo()` devuelve `False`, se sale del bucle `while` comenzado en la línea 55. Como no hay más código luego de este bucle, el programa termina.

Si `jugarDeNuevo()` devolviese `True`, la ejecución no pasaría por la sentencia `break` y regresaría a la línea 55. El programa generaría entonces un nuevo número secreto para que el jugador pudiese jugar otra vez.

## Resumen

Panecillos es un juego simple de programar pero puede ser difícil de vencer. Pero si continúas jugando, descubrirás eventualmente mejores formas de conjeturar y usar las pistas que el juego te da. De la misma forma, te convertirás en un mejor programador si continúas practicando.

Este capítulo ha introducido algunas nuevas funciones y métodos (`random.shuffle()`, `sort()` y `join()`), junto con un par de atajos. Los operadores de asignación aumentada requieren escribir menos cuando quieres cambiar el valor relativo de una variable, tal como en `spam = spam + 1`, que puede abreviarse como `spam += 1`. La interpolación de cadenas puede hacer que tu código sea mucho más legible colocando `%s` (llamado especificador de conversión) dentro de la cadena en lugar de usar muchas operaciones de concatenación de cadenas.

El siguiente capítulo no se enfoca directamente en programación, pero será necesario para los juegos que crearemos en los últimos capítulos de este libro. Aprenderemos los conceptos matemáticos de coordenadas cartesianas y números negativos. Nosotros sólo los usaremos en los juegos Sonar, Reversi y Evasor, pero estos conceptos se usan en muchos juegos. Si ya conoces estos conceptos, igual puedes hojear el siguiente capítulo para refrescarlos.

# Capítulo 12
# COORDENADAS CARTESIANAS

Temas Tratados En Este Capítulo:
- Sistemas de Coordenadas Cartesianas
- Los ejes X e Y
- La Propiedad Conmutativa de la Adición
- Valores absolutos y la función `abs()`

Este capítulo no introduce un nuevo juego, sin embargo repasa ciertos conceptos matemáticos que serán utilizados en el resto de los juegos en este libro.

Cuando miras un juego 2D (como Tetris o un viejo juego de Super Nintendo o Sega Genesis) puedes notar que la mayoría de los gráficos en la pantalla pueden moverse hacia la izquierda o derecha (la primera dimensión) o arriba y abajo (la segunda dimensión, es decir 2D). Para que podamos crear juegos con objetos moviéndose en dos dimensiones (como una pantalla de computadora bidimensional), necesitamos un sistema que pueda traducir un lugar en la pantalla a enteros que nuestro programa pueda lidiar.

Aquí es cuando se utilizan los sistemas de coordenadas Cartesianas. Las coordenadas pueden apuntar a un punto muy específico de la pantalla para que nuestro programa pueda rastrear diferentes areas en la pantalla.

## Cuadrículas y Coordenadas Cartesianas


Figura 12-1: Un ejemplo de tablero de ajedrez con un caballo negro en a, 4 y un caballo blanco en e, 6.

Un problema en muchos juegos es como hablar de puntos exactos en un tablero. Una solución común a este problema es marcar cada fila y columna individual del tablero con una letra y un número. La Figura 12-1 es un talbero de ajedrez con sus filas y columnas marcadas.

En el ajedrez, el caballo luce como una cabeza de caballo. El caballo blanco se encuentra en el punto e, 6 y el caballo negro en el a, 4. También podemos observar que todos los espacios en la fila 7 y todos los espacios en la columna c se encuentran vacios.

Una cuadrícula con filas y columnas numeradas como el tablero de ajedrez es un sistema de coordenadas cartesianas. Al utilizar una etiqueta para filas y columnas, podemos dar una coordenada para un único espacio en el tablero. Esto realmente nos ayuda a describirle a uan computadora la posición exacta que deseamos. Si aprendiste coordenadas Cartesianas en alguna clase de matemática, sabrás que usualmente se tanto las filas como columnas se representan con números. Esto es útil, porque de otro modo luego de la columna 2 nos quedaríamos sin letras. Dicho tablero se vería como la Figura 12-2.


Figura 12-2: El mismo tablero de ajedrez pero con coordenadas numéricas para filas y columnas.

Los números de izquierda a derecha que describen las columnas son **parte del eje X**. Los números de arriba a abajo que describen las filas son **parte del eje Y**. Cuando describimos una coordenada, siempre empleamos el eje X primero, seguido del eje Y. Eso significa que el caballo de la figura superior se encuentra en la coordenada 5, 6 (y no 6, 5). El caballo blanco se encuentra en la coordenada 1, 4 (no confundir con 4, 1).

Nota que para mover el caballo negro a la posición del caballo blanco, el caballo negro debe moverse dos espacios hacia arriba y luego cuatro a la derecha. (O moverse cuatro a la derecha y luego dos arriba.) Perno no necesitamos mirar el tablero para deducir esto. Si sabemos que el caballo blanco se encuentra en 5, 6 y el negro en 1, 4, entonces simplemente podemos restar para obtener la información.

Resta la coordenada X del caballo negro y la coordenada X del caballo blanco: 5 - 1 = 4. Eso significa que el caballo negro debe moverse por el eje X cuatro espacios.

Resta la coordenada Y del caballo negro y la coordenada Y del caballo blanco: 6 - 4 = 2. Eso significa que el caballo negro debe moverse por el eje Y dos espacios.

## Números Negativos

Otro concepto utilizado en las coordenadas Cartesianas son **los números negativos**. Estos son números menores a cero. Ponemos un signo menos frente al número para mostrar que es un número netagivo. -1 es menor a 0. Y -2 menor a -1. Y -3 menor a -2. Si piensas en los números regulares (llamados **positivos**) empezando del 1 e incrementando, puedes pensar en los números negativos comenzando del -1 y decreciendo. 0 en si mismo no es positivo ni negativo. En esta imagen, puedes ver los números positivos creciendo hacia la derecha y los negativos decreciendo a la izquierda:

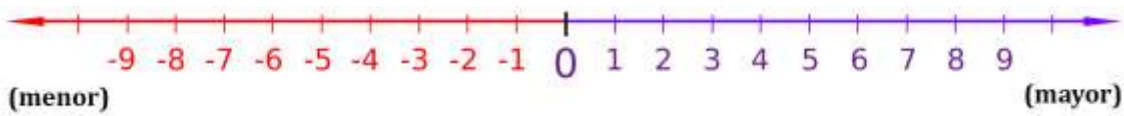



Figura 12-3: Línea de números.

La línea de números es realmente útil para realizar sumas y restas con números negativos. La expresión 4 + 3 puede ser pensada como que el caballo blanco comienza en la posición 4 y se mueve 3 espacios hacia la derecha (suma implica incrementar, es decir en la dirección derecha).

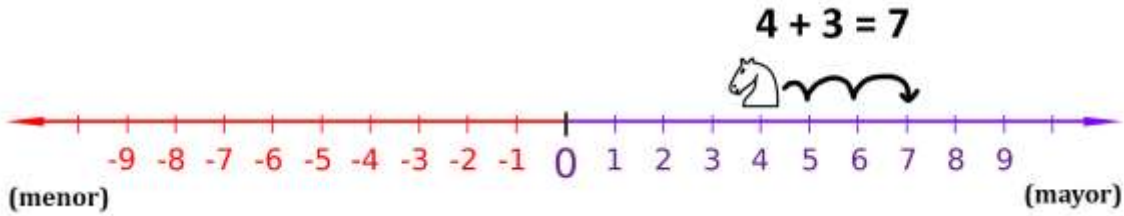



Figura 12-4: Mover el caballo blanco a la derecha suma a la coordenada.

Como puedes observar, el caballo blanco termina en la posición 7. Esto tiene sentido, porque 4 + 3 = 7.

La substracción (resta) puede realizarse moviendo el caballo blanco hacia la izquierda. Substracción implica decrementar, es decir dirección izquierda. 4 - 6 sería el caballero blanco comenzando en la posición 6 y moviéndose 6 espacios a la izquierda, como en la Figura 12-5:

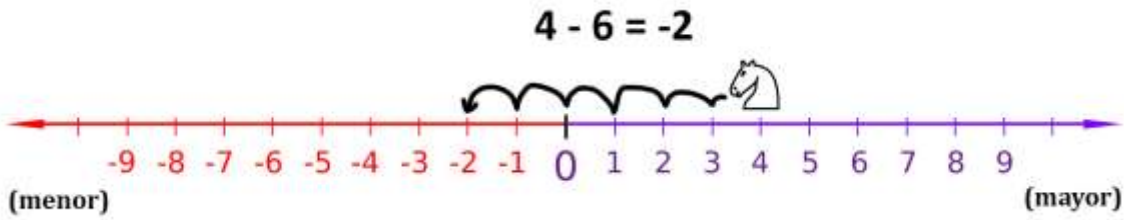



Figura 12-5: Mover el caballo blanco a la izquierda resta a la coordenada.

El caballo blanco termina en la posición -2. Eso significa 4 - 6 = -2.

Si sumamos o restamos un número negativo, el caballo blanco se moverá en direcciones *opuestas*. Si sumas un número negativo, el caballo se mueve a la *izquierda*. Si restas un número negativo, el caballo se mueve a la *derecha*. La expresión -6 - -4 sería igual a -2. El caballo comienza en -6 y se mueve a la derecha 4 espacios. Nota que -6 - -4 es lo mismo que -6 + 4.

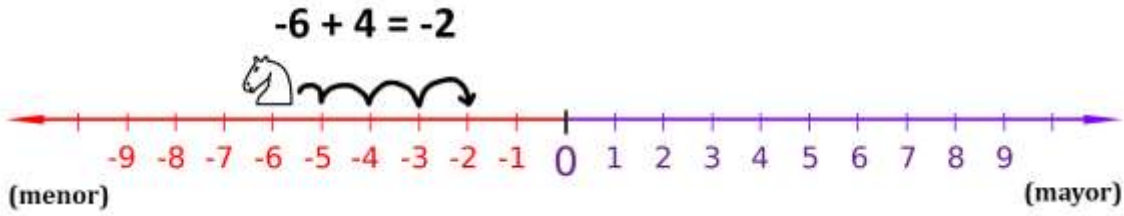



Figura 12-6: Incluso si el caballero blanco comienza en una coordenada negativa, moverse a la derecha suma a la coordenada.

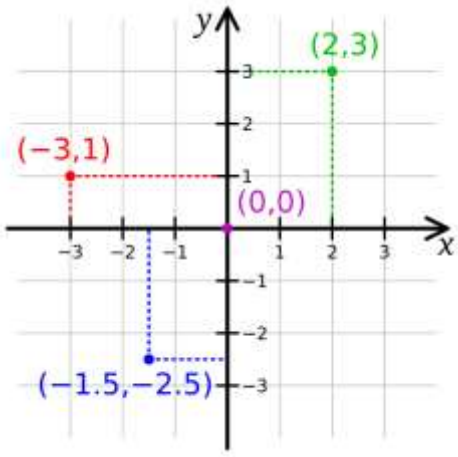



Figura 12-7: Poníendo dos líneas de números juntas se crea un sistema de coordenadas Cartesianas.

La línea de números es igual al eje-X. Si hicieramos que la línea de números vaya de arriba a abajo en vez de izquierda a derecha, sería igual al eje-Y. Sumando un número positivo (o restando un número negativo) movería el caballo hacia arriba de lal ínea, y restando un número positivo (o sumando un número negativo) movería el caballo hacia abaho. Cuando podemos ambas líneas de números juntas, tenemos un sistema de coordenadas Cartesianas tal como en la Figura 12-7. La coordenada 0, 0 posee un nombre especial: **el origen**.

## Trucos Matemáticos

Sumar o restar números negativos parece fácil cuando tienes una línea de números frente a ti, pero puede ser igual de fácil cuando sólo tienes los números. Aquí hay tres trucos que puedes hacer para que evaluar estas expresiones te sea más sencillo.

### *Truco 1: "Un Menos Come el Signo Mas a su Izquierda"*

El primer truco es si estas sumando un número negativo, por ejemplo; 4 + -2. EL primer truco es "un menos come el signo más a su izquierda". Cuando veas un signo menos con un signo más a su izquierda, puedes reemplazar el signo más con el signo menos. La respuesta es la misma, porque sumar un número negativo es lo mismo que restar un número positivo. 4 + -2 y 4 - 2 son equivalentes y dan 2.


Figura 12-8: Truco 1 - Sumando un número positivo y un número negativo.

### *Truco 2: "Dos Menos se Combinan En un Mas"*

El segundo truco es si estas restando un número negativo, por ejemplo 4 - -2. El segundo truco es "dos menos se combinan en un mas". Cuando veas dos signos menos juntos sin un número entre ellos, pueden combinarse en un signo mas. La respuesta es la misma, porque restar un valor negativo es lo mismo que sumar el mismo valor positivo.

Figura 12-9: Truco 2 - Restando un número positivo y un número negativo.

## *Truco 3: La Propiedad Conmutativa de la Adición*

El tercer truco es recordar que cuando sumas dos números como 6 y 4, no importa en que orden se encuentra. (Esto puede llamarse **la propiedad conmutativa de la adición**.) Eso significa que 6 + 4 y 4 + 6 ambos son iguales al mismo valor, 10. Si cuentas las casillas en la figura inferior, puedes ver que no importa en que orden tienes los números para sumarition.

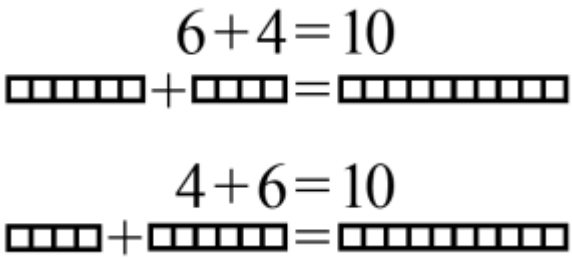



Figura 12-10: Truco 3 - La propiedad conmutativa de la adición.

Digamos que estás sumando un número negativo y un número positivo, como -6 + 8. Porque estas sumando números, puedes invertir el orden de los números sin cambiar la respuesta. -6 + 8 es lo mismo que 8 + -6.

Pero cuando miras a 8 + -6, ves que el signo menos puede comer el signo más a su izquierda, y el problema se convierte en 8 - 6 = 2. Pero esto significa que -6 + 8 ¡también es 2! Hemos reconfigurado el problema para obtener el mismo resultado, pero facilitandonos la resolución sin utilizar una calculadora o la computadora.


Figura 12-11: Usando nuestros trucos matemáticos juntos.

## Valores Absolutos y la Función abs()

**El valor absoluto** de un número es el número sin el signo négativo delante de él. Esto significa que los números positivos no cambian, pero los negativos se convierten en positivos. Por ejemplo, el valor absoluto de -4 es 4. El valor absoluto de -7 es 7. El valor absoluto de 5 (el cuál es positivo) es 5.

Podemos encontrar que tan lejos se encuentran dos elementos de una línea de números al tomar el valor absoluto de su diferencia. Imagina que el caballo blanco se encuentra en la posición 4 y el negro en la -2. Para encontrar la distancia entre ambos, debes encontrar la diferencia al restar sus posiciones y luego tomando el valor absoluto de dicho resultaod.

Esto funciona sin importar el orden de los números. -2 - 4 (esto es, menos dos menos 4) es -6, y el valor absoluto de -6 es 6. Sin embargo, 4 - -2 (esto es, cuatro menos menos 2) es 6, y el valor absoluto de 6 es 6. Utilizando el valor absoluto de la diferencia es una buena práctica para encontra la distancia entre dos puntos en una línea de números (o eje).

La función `abs()` puede ser utilizada para devolver el valor absoluto de un entero. La función `abs()` es una función incorporada, por lo que no debes importar ningún módulo para utilizarla. Pasa un entero o un valor flotante y devolvera el valor absoluto:

```
>>> abs(-5)
5
>>> abs(42)
42
>>> abs(-10.5)
10.5
```

## Sistema de Coordenadas de un Monitor de Computadora

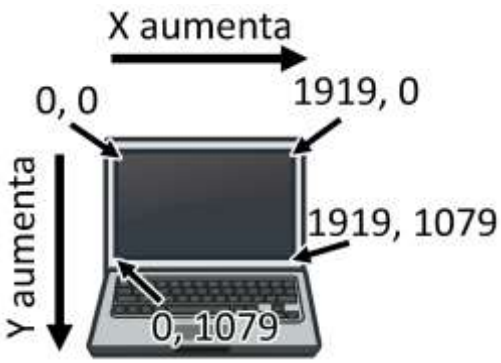



Figura 12-12: El sistema de coordenadas Cartesianas en un monitor de computadora.

Es común que los monitores de computadoras utilizen un sistema de coordenadas con origen (0, 0) en la esquina superior izquierda, el cual se incremente hacia la derecha y abajo. La mayoría de los gráficos de computadora utulizan este sistema, y lo usaremos en nuestros juegos. También es común asumir que los monitores pueden mostrar 80 caracteres por fila y 25 caracteres por columna (ver la Figura 12-12). Este solía ser el máximo tamaño de pantalla que los monitores soportaban. Mientras que los monitores actuales pueden mostrar mucho más texto, no asumiremos que la pantalla del usuario es mayor a 80 por 25.

## *Summary*

Esto no fue mucha matemática para aprender a programar. De hecho, la mayoría de la programación no requiere mucho conocimiento en matemática. Hasta este capítulo, nos las arreglabamos con simples sumas y multiplicaciones.

Los sistemas de coordenadas Cartesianas son necesarios para describir con exactitud donde se encuentra una posición en un área bidimensiona. Las coordenadas se componene de dos números: eje-X y eje-Y. El eje-X corre de izquierda a derecha y el eje-Y de arriba a abajo. En una pantalla de computadora (y la mayoría de programación), el eje-X comienza en 0 a la izquierda e incrementa hacia la derecha. El eje-Y comienza en 0 en la parte superior e incrementa hacia abajo.

Los tres trucos que aprendimos en este capítulo facilitan sumar enteros positivos y negativos. El primer truco es un signo menos que comerá un signo más a su izquierda. El segundo truco es que dos signos menos juntos se combinan en un signo más. Y el tercer truco es que puedes intercambiar las posiciones de los números que estés sumando. Esto es llamado la propiedad conmutativa de la adición.

Para el resto de este libro, utilizaremos los conceptos aprendidos en este capítulo en nuestros juegos ya que tendrán áreas bidimensionales en ellos. Todos los juegos gráficos requieren conocimientos del funcionamiento de las coordenadas Cartesianas.