Roberto Bonvallet

Apuntes de Programación

Editorial USM

Este material fue desarrollado en 2010 durante el proyecto de renovación de la asignatura *Programación de Computadores* del Departamento de Informática de la Universidad Técnica Federico Santa María, bajo la coordinación de José Miguel Herrera.

Originalmente en formato web, el apunte ha sido utilizado desde 2011 como referencia oficial de la asignatura, que es cursada por todos los estudiantes de primer año de carreras de Ingeniería de la universidad.

El contenido original del apunte está disponible libremente en la página web de la asignatura: `http://progra.usm.cl/apunte/`.

Primera impresión: enero de 2013.

ISBN: 978-956-7051-67-0.

Impresión: CIPOD Ltda. Impresión por demanda. Fono (2) 2366555.

Parte I

Materia

Capítulo 1

Introducción a la programación

> *Se suele decir que una persona no entiende algo de verdad hasta que puede explicárselo a otro. En realidad, no lo entiende de verdad hasta que puede explicárselo a un computador.* — Donald Knuth.

Si tuvieramos que resumir el propósito de la programación en una frase, ésta debería ser:

> que el computador haga el trabajo por nosotros.

Los computadores son buenos para hacer tareas rutinarias. Idealmente, cualquier problema tedioso y repetitivo debería ser resuelto por un computador, y los seres humanos sólo deberíamos encargarnos de los problemas realmente interesantes: los que requieren creatividad, pensamiento crítico y subjetividad.

La **programación** es el proceso de transformar un método para resolver problemas en uno que pueda ser entendido por el computador.

1.1 Algoritmos

> *La informática se trata de computadores tanto como la astronomía se trata de telescopios*. — Edsger Dijkstra.

Al diseñar un programa, el desafío principal es crear y describir un procedimiento que esté completamente bien definido, que no tenga ambigüedades, y que efectivamente resuelva el problema.

Así es como la programación no es tanto sobre computadores, sino sobre resolver problemas de manera estructurada. El objeto de estudio de la programación no son los programas, sino los algoritmos.

Un **algoritmo** es un procedimiento bien definido para resolver un problema.

Todo el mundo conoce y utiliza algoritmos a diario, incluso sin darse cuenta:

- Una receta de cocina es un algoritmo; si bien podríamos cuestionar que algunos pasos son ambiguos (¿cuánto es «una pizca de sal»? ¿qué significa «agregar a

gusto»?), en general las instrucciones están lo suficientemente bien definidas para que uno las pueda seguir sin problemas.

La entrada de una receta son los ingredientes y algunos datos como: ¿para cuántas personas se cocinará? El proceso es la serie de pasos para manipular los ingredientes. La salida es el plato terminado.

En principio, si una receta está suficientemente bien explicada, podría permitir preparar un plato a alguien que no sepa nada de cocina.

- El método para multiplicar números a mano que aprendimos en el colegio es un algoritmo. Dado cualquier par de números enteros, si seguimos paso a paso el procedimiento siempre obtendremos el producto.

  La entrada del algoritmo de multiplicación son los dos factores. El proceso es la secuencia de pasos en que los dígitos van siendo multiplicados las reservas van siendo sumadas, y los productos intermedios son finalmente sumados. La salida del algoritmo es el producto obtenido.

Un algoritmo debe poder ser usado mecánicamente, sin necesidad de usar inteligencia, intuición ni habilidad.

A lo largo de esta asignatura, haremos un recorrido por los conceptos elementales de la programación, con énfasis en el aspecto práctico de la disciplina.

Al final del semestre, usted tendrá la capacidad de identificar problemas que pueden ser resueltos por el computador, y de diseñar y escribir programas sencillos. Además, entenderá qué es lo que ocurre dentro del computador cuando usted usa programas.

> *Los computadores son inútiles: sólo pueden darte respuestas*. — Pablo Picasso.

Capítulo 2

Algoritmos

Un **algoritmo** es una secuencia de pasos para resolver un problema. Los pasos deben estar muy bien definidos, y tienen que describir sin ambigüedades cómo llegar desde el inicio hasta el final.

2.1 Componentes de un algoritmo

Conceptualmente, un algoritmo tiene tres componentes:

1. la **entrada**: son los datos sobre los que el algoritmo opera;
2. el **proceso**: son los pasos que hay que seguir, utilizando la entrada;
3. la **salida**: es el resultado que entrega el algoritmo.

El proceso es una secuencia de **sentencias**, que debe ser realizada en orden. El proceso también puede tener **ciclos** (grupos de sentencias que son ejecutadas varias veces) y **condicionales** (grupos de sentencias que sólo son ejecutadas bajo ciertas condiciones).

2.2 Cómo describir un algoritmo

Consideremos un ejemplo sencillo: un algoritmo para resolver ecuaciones cuadráticas.

Una ecuación cuadrática es una ecuación de la forma $ax^2 + bx + c = 0$, donde a, b y c son datos dados, con $a \neq 0$, y x es la incógnita cuyo valor que se desea determinar.

Por ejemplo, $2x^2 - 5x + 2 = 0$ es una ecuación cuadrática con $a = 2$, $b = -5$ y $c = 2$. Sus soluciones son $x_1 = 1/2$ y $x_2 = 2$, como se puede comprobar fácilmente al reemplazar estos valores en la ecuación. El problema es cómo obtener estos valores en primer lugar.

El problema computacional de resolver una ecuación cuadrática puede ser planteado así:

> Dados a, b y c, encontrar los valores reales de x que satisfacen $ax^2 + bx + c = 0$.

La entrada del algoritmo, pues, son los valores a, b y c, y la salida son las raíces reales x (que pueden ser cero, una o dos) de la ecuación. En un programa computacional, los valores de a, b y c deberían ser ingresados usando el teclado, y las soluciones x deberían ser mostradas a continuación en la pantalla.

Al estudiar álgebra aprendemos un algoritmo para resolver este problema. Es lo suficientemente detallado para que pueda usarlo cualquier persona, (incluso sin saber qué es una ecuación cuadrática) o para que lo pueda hacer un computador. A continuación veremos algunas maneras de describir el procedimiento.

Lenguaje natural

Durante el proceso mental de diseñar un algoritmo, es común pensar y describir los pasos en la misma manera en que hablamos a diario. Por ejemplo:

> Teniendo los valores de a, b y c, calcular el discriminante $\Delta = b^2 - 4ac$. Si es discriminante es negativo, entonces la ecuación no tiene soluciones reales. Si es discriminante es igual a cero, entonces la ecuación tiene una única solución real, que es $x = -b/2a$. Si el discriminante es positivo, entonces la ecuación tiene dos soluciones reales, que son $x_1 = (-b - \sqrt{\Delta})/2a$ y $x_2 = (-b + \sqrt{\Delta})/2a$.

Esta manera de expresar un algoritmo no es ideal, ya que el lenguaje natural es:

- impreciso: puede tener ambigüedades;
- no universal: personas distintas describirán el proceso de maneras distintas; y
- no estructurado: la descripción no está expresada en función de componentes simples.

Aún así, es posible identificar los pasos del algoritmo. Por ejemplo, hay que evaluar la expresión $b^2 - 4ac$, y ponerle el nombre Δ a su resultado. Esto se llama **asignación**, y es un tipo de instrucción que aparece en casi todos los algoritmos. Después de eso, el algoritmo puede usar el nombre Δ para referirse al valor calculado.

Diagrama de flujo

Un **diagrama de flujo** es una representación gráfica de un algoritmo. Los pasos son representados por varios tipos de bloques, y el flujo de ejecución es indicado por flechas que conectan los bloques, tal como se muestra en la figura 2.1.

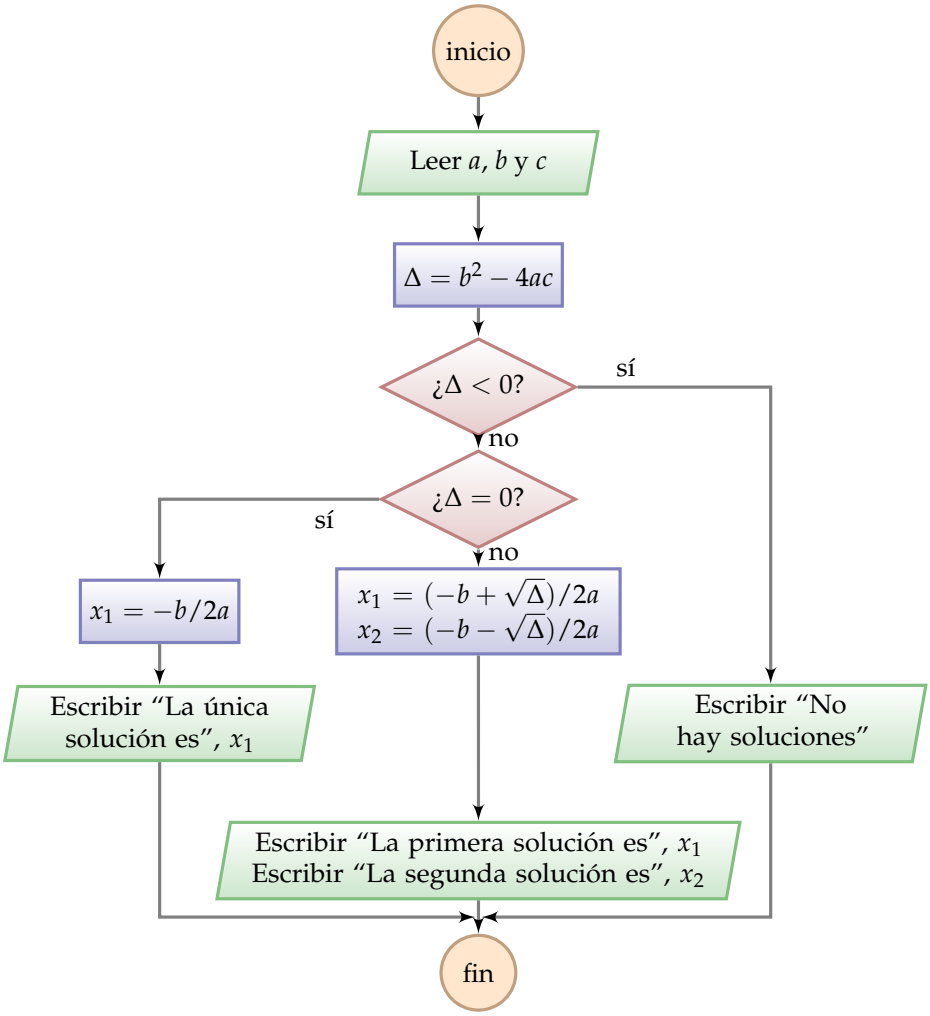


Figura 2.1: Diagrama de flujo del algoritmo para resolver la ecuación cuadrática $ax^2 + bx + c = 0$.

leer a
leer b
leer c

discriminante $= b^2 - 4ac$

si discriminante < 0:
    escribir "La ecuación no tiene soluciones reales"

o si no, si discriminante $= 0$:
    $x = -b/2a$
    escribir "La solución única es", x

o si no:
    $x_1 = (-b - \sqrt{\text{discriminante}})/2a$
    $x_2 = (-b + \sqrt{\text{discriminante}})/2a$
    escribir "La primera solución real es:", x_1
    escribir "La segunda solución real es:", x_2

Figura 2.2: Pseudocódigo del algoritmo para resolver la ecuación cuadrática $ax^2 + bx + c = 0$.

El inicio y el final del algoritmo son representados con bloques circulares. El algoritmo siempre debe ser capaz llegar desde uno hasta el otro, sin importar por qué camino lo hace. Un algoritmo no puede «quedarse pegado» en la mitad.

La entrada y la salida de datos son representadas con romboides,

Los diamantes representan condiciones en las que el algoritmo sigue uno de dos caminos que están etiquetados con *sí* o *no*, dependiendo de si la condición es verdadera o falsa.

También puede haber ciclos, representados por flechas que regresan a bloques anteriores. En este ejemplo, no hay ciclos.

Otras sentencias van dentro de rectángulos. En este ejemplo, las sentencias son asignaciones, representadas en la forma `nombre = valor`.

Los diagramas de flujo no son usados en la práctica para programar, pero son útiles para ilustrar cómo funcionan algoritmos sencillos.

Pseudocódigo

El **pseudocódigo** es una descripción estructurada de un algoritmo basada en ciertas convenciones notacionales. Si bien es muy parecido al código que finalmente se escribirá en el computador, el pseudocódigo está pensado para ser leído por humanos.

Una manera de escribir el algoritmo para la ecuación cuadrática en pseudocódigo es la que se muestra en la figura 2.2.

Las líneas que comienzan con **leer** y **escribir** denotan, respectivamente, la entrada y la salida del programa. Los diferentes casos son representados usando sentencias **si** y **o si no**. Las asignaciones siguen la misma notación que en el caso de los diagramas de flujo.

La notación de pseudocódigo es bien liberal. Uno puede mezclar notación de matemáticas y frases en español, siempre que quede absolutamente claro para el lector qué representa cada una de las líneas del algoritmo.

Código

El producto final de la programación siempre debe ser código que pueda ser ejecutado en el computador. Esto requiere describir los algoritmos en un **lenguaje de programación**. Los lenguajes de programación definen un conjunto limitado de conceptos básicos, en función de los cuales uno puede expresar cualquier algoritmo.

En esta asignatura, usaremos el lenguaje de programación Python para escribir nuestros programas.

El código en Python para resolver la ecuación cuadrática es el siguiente:

```
a = float(raw_input('Ingrese a: '))
b = float(raw_input('Ingrese b: '))
c = float(raw_input('Ingrese c: '))

discriminante = b ** 2 - 4 * a * c
if discriminante < 0:
    print 'La ecuacion no tiene soluciones reales'
elif discriminante == 0:
    x = -b / (2 * a)
    print 'La solucion unica es x =', x
else:
    x1 = (-b - (discriminante ** 0.5)) / (2 * a)
    x2 = (-b + (discriminante ** 0.5)) / (2 * a)
    print 'Las dos soluciones reales son:'
    print 'x1 =', x1
    print 'x2 =', x2

raw_input()
```

A partir de ahora, usted aprenderá a entender, escribir y ejecutar códigos como éste.

Capítulo 3

Desarrollo de programas

Un **programa** es un archivo de texto que contiene código para ser ejecutado por el computador. En el caso del lenguaje Python, el programa es ejecutado por un **intérprete**. El intérprete es un programa que ejecuta programas.

Los programas escritos en Python deben estar contenidos en un archivo que tenga la extensión `.py`.

3.1 Edición de programas

Un programa es un archivo de texto. Por lo tanto, puede ser creado y editado usando cualquier editor de texto, como el Bloc de Notas. Lo que no se puede usar es un procesador de texto como Microsoft Word.

Otros editores de texto (mucho mejores que el Bloc de Notas) que usted puede instalar son:

- en Windows: Notepad++, Textpad;
- en Mac: TextWrangler, Sublime Text, Smultron;
- en Linux: Gedit, Kate.

3.2 Instalación del intérprete de Python

Una cosa es editar el programa y otra es ejecutarlo. Para poder ejecutar un programa en Python hay que instalar el **intérprete**.

En la página de descargas de Python[1]está la lista de instaladores. Debe descargar el indicado para su computador y su sistema operativo.

La versión que será usada en la asignatura es la **2.7**, no la 3.*x*. Todo el código presentado en este apunte está escrito en Python 2.7.

No use los instaladores que dicen `x86-64` a no ser que esté seguro que su computador tiene una arquitectura de 64 bits.

[1] `http://www.python.org/download/`

3.3 Ejecución de un programa

Una vez instalado el intérprete, ya podrá ejecutar su programa. Para hacerlo, haga doble clic sobre el ícono del programa.

3.4 Uso de la consola

La ejecución de programas no es la única manera de usar el intérprete. Si uno ejecuta Python sin pasarle ningún programa, se abre la **consola** (o **intérprete interactivo**).

La consola permite ingresar un programa línea por línea. Además, sirve para evaluar expresiones y ver su resultado inmediatamente. Esto permite usarla como si fuera una calculadora.

La consola interactiva siempre muestra el símbolo `>>>`, para indicar que ahí se puede ingresar código. En todos los libros sobre Python, y a lo largo de este apunte, cada vez que aparezca un ejemplo en el que aparezca este símbolo, significa que debe ejecutarse en la consola, y no en un programa. Por ejemplo:

```
>>> a = 5
>>> a > 10
False
>>> a ** 2
25
```

En este ejemplo, al ingresar las expresiones `a > 10` y `a ** 2`, el intérprete interactivo entrega los resultados `False` y `25`.

No hay ningún motivo para tipear el símbolo `>>>` ni en un programa ni en un certamen. No es parte de la sintaxis del lenguaje.

3.5 Entornos de desarollo

En general, usar un simple editor de texto (como los mencionados arriba) para escribir programas no es la manera más eficiente de trabajar. Los **entornos de desarrollo** (también llamados *IDE*, por sus siglas en inglés) son aplicaciones que hacen más fácil la tarea de escribir programas.

Python viene con su propio entorno de desarrollo llamado **IDLE**. Este entorno ofrece una consola y un editor de texto.

Además, hay otros buenos entornos de desarrollo más avanzados para Python, como PyScripter, WingIDE 101 y PyCharm. Usted puede probarlos y usar el que más le acomode durante el semestre.

Haga la prueba: usando alguno de estos entornos, escriba el programa para resolver ecuaciones cuadráticas del capítulo anterior, guárdelo con el nombre `cuadratica.py` y ejecútelo.

Capítulo 4

Tipos de datos

Un **tipo de datos** es la propiedad de un valor que determina su dominio (qué valores puede tomar), qué operaciones se le pueden aplicar y cómo es representado internamente por el computador.

Todos los valores que aparecen en un programa tienen un tipo.

A continuación revisaremos los tipos de datos elementales de Python. Además de éstos, existen muchos otros, y más adelante aprenderemos a crear nuestros propios tipos de datos.

4.1 Números enteros

El tipo **int** (del inglés *integer*, que significa «entero») permite representar números enteros.

Los valores que puede tomar un `int` son todos los números enteros: ... `-3`, `-2`, `-1`, `0`, `1`, `2`, `3`, ...

Los números enteros literales se escriben con un signo opcional seguido por una secuencia de dígitos:

```
1570
+4591
-12
```

4.2 Números reales

El tipo **float** permite representar números reales.

El nombre `float` viene del término *punto flotante*, que es la manera en que el computador representa internamente los números reales.

Hay que tener cuidado, porque los números reales no se pueden representar de manera exacta en un computador. Por ejemplo, el número decimal 0.7 es representado internamente por el computador mediante la aproximación 0.6999999999999996. Todas las operaciones entre valores `float` son aproximaciones. Esto puede conducir a resultados algo sorpresivos:

```
>>> 1/7 + 1/7 + 1/7 + 1/7 + 1/7 + 1/7 + 1/7
0.9999999999999998
```

Los números reales literales se escriben separando la parte entera de la decimal con un punto. Las partes entera y decimal pueden ser omitida si alguna de ellas es cero:

```
>>> 881.9843000
881.9843
>>> -3.14159
-3.14159
>>> 1024.
1024.0
>>> .22
0.22
```

Otra representación es la notación científica, en la que se escribe un factor y una potencia de diez separados por una letra `e`. Por ejemplo:

```
>>> -2.45E4
-24500.0
>>> 7e-2
0.07
>>> 6.02e23
6.02e+23
>>> 9.1094E-31
9.1094e-31
```

Los dos últimos valores del ejemplo son, respectivamente, $6{,}02 \times 10^{23}$ (la constante de Avogadro) y $9{,}1094 \times 10^{-31}$ (la masa del electrón en kilogramos).

4.3 Números complejos

El tipo **complex** permite representar números complejos. Los números complejos tienen una parte real y una imaginaria. La parte imaginaria es denotada agregando una `j` pegada inmediatamente después de su valor:

```
3 + 9j
-1.4 + 2.7j
```

4.4 Valores lógicos

Los valores lógicos **`True`** y **`False`** (verdadero y falso) son de tipo **bool**, que representa valores lógicos.

El nombre **`bool`** viene del matemático George Boole, quien creó un sistema algebraico para la lógica binaria. Por lo mismo, a **`True`** y **`False`** también se les llama **valores booleanos**. El nombre no es muy intuitivo, pero es el que se usa en informática, así que hay que conocerlo.

4.5 Texto

A los valores que representan texto se les llama **strings**, y tienen el tipo **str**. Los strings literales pueden ser representados con texto entre comillas simples o comillas dobles:

```
"ejemplo 1"
'ejemplo 2'
```

La ventaja de tener dos clases de comillas es que se puede usar una de ellas cuando la otra aparece como parte del texto:

```
"Let's go!"
'Ella dijo "hola"'
```

Es importante entender que los strings no son lo mismo que los valores que en él pueden estar representados:

```
>>> 5 == '5'
False
>>> True == 'True'
False
```

Los strings que difieren en mayúsculas y minúsculas o en espacios en blanco también son distintos:

```
>>> 'mesa' == 'Mesa'
False
>>> ' mesa' == 'mesa '
False
```

4.6 Nulo

Existe un valor llamado **None** (en inglés, «ninguno») que es utilizado para representar casos en que ningún valor es válido, o para indicar que una variable todavía no tiene un valor que tenga sentido. El valor `None` tiene su propio tipo, llamado `NoneType`, que es diferente al de todos los demás valores.

Capítulo 5

Programas simples

Un programa es una secuencia de **sentencias**. Una sentencia representa una instrucción bien definida que es ejecutada por el computador. En Python, cada línea del código representa una sentencia. Hay que distinguir entre:

1. **sentencias simples**: son una única instrucción; y
2. **sentencias de control**: contienen varias otras sentencias, que a su vez pueden ser simples o de control.

Las sentencias simples son ejecutadas secuencialmente, una después de la otra.

Todas las sentencias siguen ciertas reglas acerca de cómo deben ser escritas. Si no son seguidas, el programa está incorrecto y no se ejecutará. A este conjunto de reglas se le denomina **sintaxis**.

A continuación veremos algunas sentencias simples, con las que se pueden escribir algunos programas sencillos. Más adelante introduciremos las sentencias de control.

Como ejemplo, consideremos el siguiente programa, que pide al usuario ingresar una temperatura en grados Fahrenheit y entrega como resultado el equivalente en grados Celsius:

```
f = float(raw_input('Ingrese temperatura en Fahrenheit: '))
c = (f - 32.0) * (5.0 / 9.0)
print 'El equivalente en Celsius es:', c
```

Escriba este programa en el computador y ejecútelo para convencerse de que funciona correctamente.

5.1 Expresiones y variables

Una **expresión** es una combinación de valores y operaciones que son evaluados durante la ejecución del algoritmo para obtener un resultado.

Por ejemplo, `2 + 3` es una expresión aritmética que, al ser evaluada, siempre entrega el valor `5` como resultado. En esta expresión, `2` y `3` son **valores literales** y `+` es el operador de adición.

En nuestro programa de conversión de temperaturas aparece la expresión `(f - 32.0) * (5.0 / 9.0)`, cuyo resultado depende de cuál es el valor de `f` al momento de la evaluación. A diferencia de los valores literales, `f` es una **variable**, que tiene un valor específico que puede ser distinto cada vez que la expresión es evaluada.

En esta expresión, `*` es el operador de multiplicación y `/` el de división.

Una expresión puede ser usada como una sentencia de un programa por sí sola, pero la mayoría de las veces esto no tiene ningún efecto. El programa evaluará la expresión, pero no hará nada con el resultado obtenido.

5.2 Asignaciones

Una **asignación** es una sentencia que asocia un nombre al resultado de una expresión. El nombre asociado al valor se llama **variable**.

La sintaxis de una asignación es la siguiente:

```
variable = expresion
```

Por ejemplo, el programa de conversión de temperaturas tiene la siguiente asignación:

```
c = (f - 32.0) * (5.0 / 9.0)
```

Cuando aparece una asignación en un programa, es interpretada de la siguiente manera:

1. primero la expresión a la derecha del signo = es evaluada, utilizando los valores que tienen en ese momento las variables que aparecen en ella;
2. una vez obtenido el resultado, el valor de la variable a la izquierda del signo = es reemplazado por ese resultado.

Bajo esta interpretación, es perfectamente posible una asignación como ésta:

```
i = i + 1
```

Primero la expresión `i + 1` es evaluada, entregando como resultado el sucesor del valor actual de `i`. A continuación, la variable `i` toma el nuevo valor. Por ejemplo, si `i` tiene el valor 15, después de la asignación tendrá el valor 16.

Esto no significa que $15 = 16$. Una asignación no es una igualdad matemática ni una ecuación.

Por ejemplo, las siguientes asignaciones son correctas, suponiendo que las variables que aparecen en ellas ya fueron asignadas previamente:

```
nombre = 'Perico Los Palotes'
discriminante = b ** 2 - 4 * a * c
pi = 3.14159
```

```
r = 5.0
perimetro = 2 * pi * r
sucesor = n + 1
a = a
es_menor = x < 4
x0 = x1 + x2
r = 2 * abs(x - x0)
nombre = raw_input('Ingrese su nombre')
```

En contraste, las siguientes no son asignaciones válidas, pues no respetan la sintaxis `nombre = expresion`:

```
n + 1 = 5
7 = a
2_pi_r = 2 * pi * r
area del circulo = pi * r ** 2
x ** 2 = x * x
```

¡Identifique los errores!

5.3 Entrada

La **entrada** es la parte del programa que pide datos al usuario.

La manera más simple de ingresar datos es hacerlo a través del teclado. La función `raw_input(mensaje)` pide al usuario ingresar un valor, que puede ser asignado a una variable para ser usado más adelante por el programa. El `mensaje` es lo que se mostrará al usuario antes de que él ingrese el valor.

El valor ingresado por el usuario siempre es interpretado como texto, por lo que es de tipo `str`. Si es necesario usarlo como si fuera de otro tipo, hay que convertirlo explícitamente.

Por ejemplo, en el programa de conversión de temperaturas, la entrada es realizada por la sentencia:

```
f = float(raw_input('Ingrese temperatura en Fahrenheit: '))
```

Cuando el programa llega a esta línea, el mensaje «`Ingrese temperatura en Fahrenheit:`» es mostrado al usuario, que entonces debe ingresar un valor, que es convertido a un número real y asociado al nombre `f`.

Desde esa línea en adelante, la variable `f` puede ser usada en el programa para referirse al valor ingresado.

5.4 Salida

La **salida** es la parte del programa en que los resultados son entregados al usuario.

La manera más simple de entregar la salida es mostrando texto en la pantalla. En Python, la salida del programa es realizada por la sentencia **print** (*imprimir* en inglés).

Si se desea imprimir un texto tal cual, la sintaxis es la siguente:

```
print valor_a_imprimir
```

Si los valores a imprimir son varios, deben ser puestos separados por comas. Por ejemplo, el programa de conversión de temperaturas tiene la siguiente sentencia de salida:

```
print 'El equivalente en Celsius es:', c
```

El programa imprimirá el mensaje «`El equivalente en Celsius es:`» y a continuación, en la misma línea, el valor de la variable `c`.

Las comillas sólo sirven para representar un string en el código, y no forman parte del string. Al imprimir el string usando **`print`** las comillas no aparecen:

```
>>> 'Hola'
'Hola'
>>> print 'Hola'
Hola
```

5.5 Comentarios

Un **comentario** es una sección del código que es ignorada por el intérprete. Un comentario puede ser utilizado por el programador para dejar un mensaje en el código que puede ser útil para alguien que tenga que leerlo en el futuro.

En Python, cualquier texto que aparezca a la derecha de un signo `#` es un comentario:

```
>>> 2 + 3  # Esto es una suma
5
>>> # Esto es ignorado
>>>
```

La excepción son los signos `#` que aparecen en un string:

```
>>> "123 # 456" # 789
'123 # 456'
```

5.6 Evitar que se cierre el programa

La ejecución de programas en Windows presenta un inconveniente práctico: cuando el programa termina, la ventana de ejecución se cierra inmediatamente, por lo que no es posible alcanzar a leer la salida del programa.

Por ejemplo, al ejecutar el programa `temperatura.py` tal como está al comienzo del capítulo, el usuario verá el mensaje «`Ingrese temperatura...`» y a continuación ingresará el valor. Una vez que el programa entrega como resultado el equivalente en grados Celsius, no quedan más sentencias para ejecutar, por lo que el programa se cierra.

Existen otras maneras de ejecutar programas con las que este problema no ocurre. Por ejemplo, al ejecutar un programa desde una IDE, generalmente la salida aparece en una ventana que no se cierra.

Una solución para evitar que la ventana se cierre es agregar un `raw_input()` al final del código. De este modo, el programa quedará esperando que el usuario ingrese cualquier cosa (un enter basta) antes de cerrarse.

Los programas presentados en este apunte no tendrán el `raw_input()` al final, pero usted puede agregarlo por su cuenta si así lo desea. En controles y certámenes, no será necesario hacerlo.

Capítulo 6

Expresiones

Una **expresión** es una combinación de valores y operaciones que, al ser evaluados, entregan un valor como resultado.

Algunos elementos que pueden formar parte de una expresión son: valores **literales** (como `2`, `"hola"` o `5.7`), **variables**, **operadores** y **llamadas a funciones**.

Por ejemplo, la expresión `4 * 3 - 2` entrega el valor 10 al ser evaluada por el intérprete:

```
>>> 4 * 3 - 2
10
```

El valor de la siguiente expresión depende del valor que tiene la variable `n` en el momento de la evaluación:

```
>>> n / 7 + 5
```

Una expresión está compuesta de otras expresiones, que son evaluadas recursivamente hasta llegar a sus componentes más simples, que son los literales y las variables.

6.1 Operadores

Un **operador** es un símbolo en una expresión que representa una operación aplicada a los valores sobre los que actúa.

Los valores sobre los que actúa un operador se llaman **operandos**. Un **operador binario** es el que tiene dos operandos, mientras que un **operador unario** es el que tiene sólo uno.

Por ejemplo, en la expresión `2.0 + x` el operador `+` es un operador binario que en este contexto representa la operación de adición. Sus operandos son `2.0` y `x`.

Las operaciones más comunes se pueden clasificar en: aritméticas, relacionales, lógicas y de texto.

Operadores aritméticos

Las **operaciones aritméticas** son las que operan sobre valores numéricos y entregan otro valor numérico como resultado. Los valores numéricos son los que tienen tipo entero, real o complejo.

Las siguientes son algunas operaciones aritméticas básicas, junto con el operador que las representa en Python:

- la **suma** `+`;
- la **resta** `-`;
- la **multiplicación** `*`;
- la **división** `/`;
- el **módulo** `%` (resto de la división);
- la **potencia** `**` («elevado a»).

En general, si los operandos son de tipo entero, el resultado también será de tipo entero. Pero basta que uno de los operandos sea real para que el resultado también lo sea:

```
>>> 8 - 5
3
>>> 8 - 5.0
3.0
>>> 8.0 - 5
3.0
>>> 8.0 - 5.0
3.0
```

Esta regla suele causar confusión en el caso de la división. Al dividir números enteros, el resultado siempre es entero, y es igual al resultado real **truncado**, es decir, sin su parte decimal:

```
>>> 5 / 2
2
>>> 5 / -2
-3
```

Si uno de los operandos es complejo, el resultado también será complejo:

```
>>> 3 + 4
7
>>> 3 + (4+0j)
(7+0j)
```

El operador de módulo entrega el resto de la división entre sus operandos:

```
>>> 7 % 3
1
```

El operador de módulo es usado comúnmente para determinar si un número es divisible por otro:

```
>>> 17 % 5   # 17 no es divisible por 5
2
>>> 20 % 5   # 20 si es divisible por 5
0
```

Una relación entre `/` y `%` que siempre se cumple para los números enteros es:

```
(a / b) * b + (a % b) == a
```

Hay dos operadores aritméticos unarios: el **positivo** +, y el **negativo** −. El positivo entrega el mismo valor que su operando, y el negativo también pero con el signo cambiado:

```
>>> n = -4
>>> +n
-4
>>> -n
4
```

Operaciones relacionales

Las **operaciones relacionales** sirven para comparar valores. Sus operandos son cualquier cosa que pueda ser comparada, y sus resultados siempre son valores lógicos. Algunas operaciones relacionales son:

- el **igual a** == (no confundir con el = de las asignaciones);
- el **distinto a** !=;
- el **mayor que** >;
- el **mayor o igual que** >=;
- el **menor que** <;
- el **menor o igual que** <=;

Algunos ejemplos en la consola:

```
>>> a = 5
>>> b = 9
>>> c = 14
>>> a < b
True
```

p	q	p **and** q	p **or** q	**not** p
True	**True**	**True**	**True**	**False**
True	**False**	**False**	**True**	
False	**True**	**False**	**True**	**True**
False	**False**	**False**	**False**	

Tabla 6.1: Resultados posibles de los operadores booleanos.

```
>>> a + b != c
False
>>> 2.0 == 2
True
>>> 'amarillo' < 'negro'
True
```

Los operadores relacionales pueden ser encadenados, tal como se acostumbra en matemáticas, de la siguiente manera:

```
>>> x = 4
>>> 0 < x <= 10
True
>>> 5 <= x <= 20
False
```

La expresión `0 < x <= 10` es equivalente a `(0 < x) and (x <= 10)`.

Operaciones lógicas

Los **operadores lógicos** son los que tienen operandos y resultado de tipo lógico. En Python hay tres operaciones lógicas:

- la conjunción lógica **and** (en español: «y»),
- la disyunción lógica **or** (en español: «o»), y
- la negación lógica **not** (en español: «no»).

Los operadores **and** y **or** son binarios, mientras que **not** es unario:

```
>>> True and False
False
>>> not True
False
```

La tabla 6.1 muestra todos los resultados posibles de las operaciones lógicas. Las primeras dos columnas representan los valores de los operandos, y las siguientes tres, los resultados de las operaciones.

Operaciones de texto

Los operadores + y * tienen otras interpretaciones cuando sus operandos son strings. + es el operador de **concatenación** de strings: pega dos strings uno después del otro:

```
>>> 'perro' + 'gato'
'perrogato'
```

La concatenación no es una suma. Ni siquiera es una operación conmutativa.

* es el operador de **repetición** de strings. Recibe un operando string y otro entero, y entrega como resultado el string repetido tantas veces como indica el entero:

```
>>> 'waka' * 2
'wakawaka'
```

Más adelante veremos muchas más operaciones para trabajar sobre texto. Por ahora utilizaremos las más elementales. Otras operaciones que pueden serle útiles por el momento son:

- obtener el *i*-ésimo caracter de un string (partiendo desde cero) usando los corchetes:

```
>>> nombre = 'Perico'
>>> nombre[0]
'P'
>>> nombre[1]
'e'
>>> nombre[2]
'r'
```

- comprarar strings alfabéticamente con los operadores relacionales (lamentablemente no funciona con acentos y eñes):

```
>>> 'a' < 'abad' < 'abeja'
True
>>> 'zapato' <= 'alpargata'
False
```

- obtener el largo de un string con la función `len`:

```
>>> len('papalelepipedo')
14
>>> len("")
0
```

- verificar si un string está dentro de otro con el operador `in`:

```
>>> 'pollo' in 'repollos'
True
>>> 'pollo' in 'gallinero'
False
```

6.2 Precedencia

La **precedencia de operadores** es un conjunto de reglas que especifica en qué orden deben ser evaluadas las operaciones de una expresión.

La precedencia está dada por la siguiente lista, en que los operadores han sido listados en orden de menor a mayor precedencia:

- **or**;
- **and**;
- **not**;
- <, <=, >, >=, !=, ==;
- +, − (suma y resta);
- *, /, %;
- +, − (positivo y negativo);
- **.

Esto significa, por ejemplo, que las multiplicaciones se evalúan antes que las sumas, y que las comparaciones se evalúan antes que las operaciones lógicas:

```
>>> 2 + 3 * 4
14
>>> 1 < 2 and 3 < 4
True
```

Operaciones dentro de un mismo nivel son evaluadas en el orden en que aparecen en la expresión, de izquierda a derecha:

```
>>> 15 * 12 % 7    # es igual a (15 * 12) % 7
5
```

La única excepción a la regla anterior son las potencias, que son evaluadas de derecha a izquierda:

```
>>> 2 ** 3 ** 2    # es igual a 2 ** (3 ** 2)
512
```

Para forzar un orden de evaluación distinto a la regla de precedencia, debe usarse paréntesis:

```
>>> (2 + 3) * 4
20
>>> 15 * (12 % 7)
75
>>> (2 ** 3) ** 2
64
```

Otra manera de forzar el orden es ir guardando los resultados intermedios en variables:

```
>>> n = 12 % 7
>>> 15 * n
75
```

Como ejemplo, consideremos la siguiente expresión:

```
15 + 59 * 75 / 9 < 2 ** 3 ** 2 and (15 + 59) * 75 % n == 1
```

y supongamos que la variable `n` tiene el valor 2. Aquí podemos ver cómo la expresión es evaluada hasta llegar al resultado final, que es **`False`**:

```
15 + 59 * 75 / 9 < 2 ** 3 ** 2 and (15 + 59) * 75 % n == 1
15 + 59 * 75 / 9 < 2 **   9    and (15 + 59) * 75 % n == 1
15 + 59 * 75 / 9 < 512         and (15 + 59) * 75 % n == 1
15 +  4425   / 9 < 512         and (15 + 59) * 75 % n == 1
15 +        491  < 512         and (15 + 59) * 75 % n == 1
15 +        491  < 512         and    74     * 75 % n == 1
15 +        491  < 512         and          5550  % n == 1
15 +        491  < 512         and          5550  % 2 == 1
15 +        491  < 512         and                0   == 1
  506           < 512         and                0   == 1
              True             and                0   == 1
              True             and                  False
                               False
```

La operación entre paréntesis `(15 + 59)` debe ser evaluada antes de la multiplicación por 75, ya que es necesario conocer su resultado para poder calcular el producto. El momento preciso en que ello ocurre no es importante.

Lo mismo ocurre con la evaluación de la variable `n`: sólo importa que sea evaluada antes de ser usada por el operador de módulo.

En el ejemplo, ambos casos fueron evaluados inmediatamente antes de que su valor sea necesario.

Las reglas completas de precedencia, incluyendo otros operadores que aún no hemos visto, pueden ser consultados en la sección sobre expresiones de la documentación oficial de Python.

¿Cómo aprenderse las reglas de precedencia?

La respuesta es: mejor no aprendérselas. Las reglas de precedencia son muchas y no siempre son intuitivas.

Un programa queda mucho más fácil de entender si uno explícitamente indica el orden de evaluación usando paréntesis o guardando en variables los resultados intermedios del cálculo.

Un buen programador siempre se preocupa de que su código sea fácil de entender por otras personas, ¡e incluso por él mismo en unas semanas más adelante!

6.3 Llamadas a función

Los operadores forman un conjunto bastante reducido de operaciones. Más comúnmente, las operaciones más generales son representadas como **funciones**.

Al igual que en matemáticas, las funciones tienen un nombre, y reciben **parámetros** (o **argumentos**) que van entre paréntesis después del nombre. La operación de usar la función para obtener un resultado se llama **llamar la función**.

Ya conocemos la función `raw_input`, que entrega como resultado el texto ingresado por el usuario mediante el teclado.

La función `abs` entrega el valor absoluto de su argumento:

```
>>> abs(4 - 5)
1
>>> abs(5 - 4)
1
```

La función `len` recibe un string y entrega su largo:

```
>>> len('hola mundo')
10
>>> len('hola' * 10)
40
```

Los nombres de los tipos también sirven como funciones, que entregan el equivalente de su parámetro en el tipo correspondiente:

```
>>> int(3.8)
3
>>> float('1.5')
1.5
>>> str(5 + 6)
'11'
>>> int('5' + '6')
56
```

Las funciones `min` y `max` entregan el mínimo y el máximo de sus argumentos:

```
>>> min(6, 1, 8)
1
```

```
>>> min(6.0, 1.0, 8.0)
1.0
>>> max(6, 1, 4, 8)
8
```

La función `round` redondea un número real al entero más cercano:

```
>>> round(4.4)
4.0
>>> round(4.6)
5.0
```

Algunas funciones matemáticas como la exponencial, el logaritmo y las trigonométricas pueden ser usadas, pero deben ser importadas antes usando la sentencia **`import`**, que veremos en detalle más adelante:

```
>>> from math import exp
>>> exp(2)
7.3890560989306504
>>> from math import sin, cos
>>> cos(3.14)
-0.9999987317275395
>>> sin(3.14)
0.0015926529164868282
```

La lista completa de funciones matemáticas que pueden ser importadas está en la descripción del módulo `math` en la documentación de Python.

Más adelante también aprenderemos a crear nuestras propias funciones. Por ahora, sólo necesitamos saber cómo llamarlas.

Por supuesto, siempre es necesario que los argumentos de una llamada tengan el tipo apropiado:

```
>>> round('perro')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: a float is required
>>> len(8)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Capítulo 7

Errores y excepciones

No siempre los programas que escribiremos están correctos. Existen muchos tipos de errores que pueden estar presentes en un programa.

No todos los errores pueden ser detectados por el computador. Por ejemplo, el siguiente programa tiene un error lógico bastante evidente:

```
n = int(raw_input('Escriba un numero: '))
doble = 3 * n
print 'El doble de n es', doble
```

El computador no se dará cuenta del error, pues todas las instrucciones del programa son correctas. El programa simplemente entregará siempre la respuesta equivocada.

Existen otros errores que sí pueden ser detectados. Cuando un error es detectado *durante* la ejecución del programa ocurre una **excepción**.

El intérprete anuncia una excepción deteniendo el programa y mostrando un mensaje con la descripción del error. Por ejemplo, podemos crear el siguiente programa y llamarlo `division.py`:

```
n = 8
m = 0
print n / m
print 'Listo'
```

Al ejecutarlo, el intérprete lanzará una excepción, pues la división por cero es una operación inválida:

```
Traceback (most recent call last):
  File "division.py", line 3, in <module>
    print n / m
ZeroDivisionError: division by zero
```

La segunda línea del mensaje indica cómo se llama el archivo donde está el error y en qué línea del archivo está. En este ejemplo, el error esta en la línea 3 de `division.py`. La última línea muestra el nombre de la excepción (en este caso es `ZeroDivisionError`) y un mensaje explicando cuál es el error.

Los errores y excepciones presentados aquí son los más básicos y comunes.

7.1 Error de sintaxis

Un **error de sintaxis** ocurre cuando el programa no cumple las reglas del lenguaje. Cuando ocurre este error, significa que el programa está mal escrito. El nombre del error es `SyntaxError`.

Los errores de sintaxis siempre ocurren *antes* de que el programa sea ejecutado. Es decir, un programa mal escrito no logra ejecutar ninguna instrucción. Por lo mismo, el error de sintaxis no es una excepción.

A continuación veremos algunos ejemplos de errores de sintaxis

```
>>> 2 * (3 + 4))
  File "<stdin>", line 1
    2 * (3 + 4))
               ^
SyntaxError: invalid syntax

>>> n + 2 = 7
  File "<stdin>", line 1
SyntaxError: can't assign to operator

>>> True = 1000
  File "<stdin>", line 1
SyntaxError: assignment to keyword
```

7.2 Error de nombre

Un **error de nombre** ocurre al usar una variable que no ha sido creada con anterioridad. El nombre de la excepción es `NameError`:

```
>>> x = 20
>>> 5 * x
100
>>> 5 * y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

Para solucionar este error, es necesario asignar un valor a la variable antes de usarla.

7.3 Error de tipo

En general, todas las operaciones en un programa pueden ser aplicadas sobre valores de tipos bien específicos. Un **error de tipo** ocurre al aplicar una

operación sobre operandos de tipo incorrecto. El nombre de la excepción es `TypeError`.

Por ejemplo, no se puede multiplicar dos strings:

```
>>> 'seis' * 'ocho'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

Tampoco se puede obtener el largo de un número:

```
>>> len(68)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Cuando ocurre un error de tipo, generalmente el programa está mal diseñado. Hay que revisarlo, idealmente hacer un ruteo para entender el error, y finalmente corregirlo.

7.4 Error de valor

El **error de valor** ocurre cuando los operandos son del tipo correcto, pero la operación no tiene sentido para ese valor. El nombre de la excepción es `ValueError`.

Por ejemplo, la función `int` puede convertir un string a un entero, pero el string debe ser la representación de un número entero. Cualquier otro valor lanza un error de valor:

```
>>> int('41')
41
>>> int('perro')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'perro'
>>> int('cuarenta y uno')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10:
                                              'cuarenta y uno'
```

Para corregir el error, hay que preocuparse de siempre usar valores adecuados.

7.5 Error de división por cero

El **error de division por cero** ocurre al intentar dividir por cero. El nombre de la excepción es `ZeroDivisionError`:

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

7.6 Error de desborde

El **error de desborde** ocurre cuando el resultado de una operación es tan grande que el computador no puede representarlo internamente. El nombre de la excepción es `OverflowError`:

```
>>> 20.0 ** 20.0 ** 20.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')
```

Los interesados en profundizar más pueden revisar la sección sobre excepciones en la documentación oficial de Python.

Capítulo 8

Sentencias de control

Un programa es una sucesión de **sentencias** que son ejecutadas secuencialmente. Por ejemplo, el siguiente programa tiene cuatro sentencias:

```
n = int(raw_input('Ingrese n: '))
m = int(raw_input('Ingrese m: '))
suma = n + m
print 'La suma de n y m es:', suma
```

Las primeras tres son asignaciones, y la última es una llamada a función. Al ejecutar el programa, cada una de estas sentencias es ejecutada, una después de la otra, una sola vez.

Además de las sentencias simples, que son ejecutadas en secuencia, existen las **sentencias de control** que permiten modificar el flujo del programa introduciendo ciclos y condicionales.

Un **condicional** es un conjunto de sentencias que pueden o no ejecutarse, dependiendo del resultado de una condición.

Un **ciclo** es un conjunto de sentencias que son ejecutadas varias veces, hasta que una condición de término es satisfecha.

Tanto los condicionales como los ciclos contienen a otras sentencias. Para indicar esta relación se utiliza la **indentación**: las sentencias contenidas no se escriben en la misma columna que la sentencia de control, sino un poco más a la derecha:

```
n = int(raw_input())
m = int(raw_input())
if m < n:
    t = m
    m = n
    n = t
print m, n
```

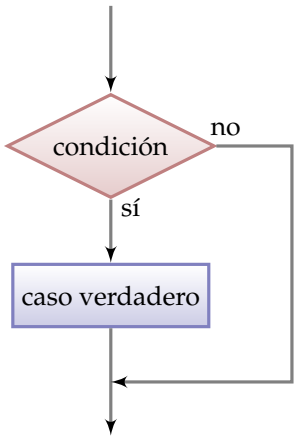
En este ejemplo, las tres asignaciones están contenidas dentro de la sentencia de control `if`. El `print m, n` no está indentado, por lo que no es parte de la sentencia `if`.

Este programa tiene cuatro sentencias, de las cuales la tercera es una sentencia de control, que contiene a otras tres sentencias.

Para indentar, utilizaremos siempre cuatro espacios.

8.1 Condicional if

La sentencia **if** («si») ejecuta las instrucciones sólo si se cumple una condición. Si la condición es falsa, no se hace nada:



La sintaxis es la siguiente:

```
if condicion:
    sentencias
```

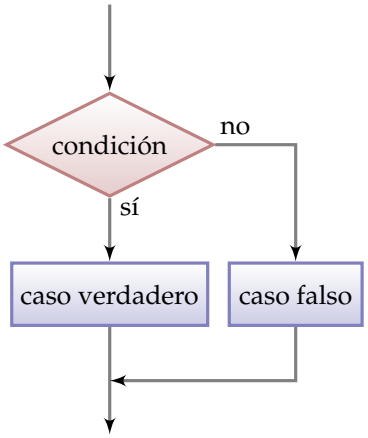
Por ejemplo, el siguente programa felicita a alguien que aprobó el ramo:

```
nota = int(raw_input('Ingrese su nota: '))
if nota >= 55:
    print 'Felicitaciones'
```

Ejecute este programa, probando varias veces con valores diferentes.

8.2 Condicional if-else

La sentencia **if-else** («si-o-si-no») decide qué instrucciones ejecutar dependiendo si una condición es verdadera o falsa:



La sintaxis es la siguiente:

```
if condicion:
    que hacer cuando la condicion es verdadera
else:
    que hacer cuando la condicion es falsa
```

Por ejemplo, el siguiente programa indica a alguien si es mayor de edad:

```
edad = int(raw_input('Cual es su edad? '))
if edad < 18:
    print 'Usted es menor de edad'
else:
    print 'Usted es adulto'
```

El siguiente programa realiza acciones distintas dependiendo de si el número de entrada es par o impar:

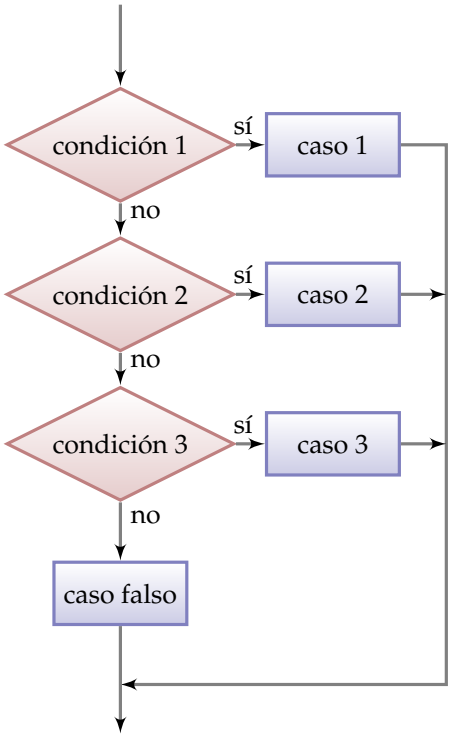
```
n = int(raw_input('Ingrese un numero: '))
if n % 2 == 0:
    print 'El numero es par'
    print 'La mitad del numero es', n / 2
else:
    print 'El numero es impar'
    print 'El sucesor del numero es', n + 1
print 'Listo'
```

La última sentencia no está indentada, por lo que no es parte del condicional, y será ejecutada siempre.

8.3 Condicional if-elif-else

La sentencia **if-elif-else** depende de dos o más condiciones, que son evaluadas en orden. La primera que es verdadera determina qué instrucciones

serán ejecutadas:



La sintaxis es la siguiente:

```
if condicion1:
    que hacer si condicion1 es verdadera
elif condicion2:
    que hacer si condicion2 es verdadera
...
else:
    que hacer cuando ninguna de las
    condiciones anteriores es verdadera
```

El último **else** es opcional.

Por ejemplo, la tasa de impuesto a pagar por una persona según su sueldo puede estar dada por la tabla 8.1. Entonces, el programa que calcula el impuesto a pagar es el siguiente:

```
sueldo = int(raw_input('Ingrese su sueldo: '))
if sueldo < 1000:
    tasa = 0.00
elif sueldo < 2000:
```

Sueldo	Tasa de impuesto
menos de 1000	0 %
$1000 \leq$ sueldo < 2000	5 %
$2000 \leq$ sueldo < 4000	10 %
4000 o más	12 %

Tabla 8.1: Ejemplo: tasa de impuesto en función del sueldo recibido.

```
    tasa = 0.05
elif sueldo < 4000:
    tasa = 0.10
else:
    tasa = 0.12
print 'Usted debe pagar', tasa * sueldo, 'de impuesto'
```

Siempre sólo una de las alternativas será ejecutada. Tan pronto alguna de las condiciones es verdadera, el resto de ellas no siguen siendo evaluadas.

Otra manera de escribir el mismo programa usando sólo sentencias **if** es la siguiente:

```
sueldo = int(raw_input('Ingrese su sueldo: '))
if sueldo < 1000:
    tasa = 0.00
if 1000 <= sueldo < 2000:
    tasa = 0.05
if 2000 <= sueldo < 4000:
    tasa = 0.10
if 4000 < sueldo:
    tasa = 0.12
print 'Usted debe pagar', tasa * sueldo, 'de impuesto'
```

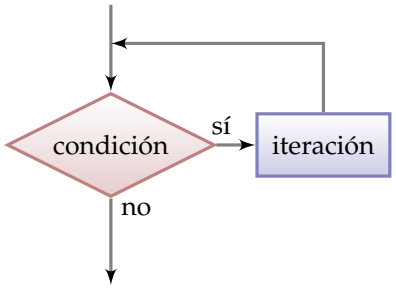
Esta manera es menos clara, porque no es evidente a primera vista que sólo una de las condiciones será verdadera.

8.4 Ciclo while

El ciclo **while** («mientras») ejecuta una secuencia de instrucciones mientras una condición sea verdadera:

p	m	n
	4	
		7
0		
	3	
7		
	2	
14		
	1	
21		
	0	
28		

Tabla 8.2: Ruteo del programa de ejemplo. La tabla muestra cómo cambian los valores de cada variable durante la ejecución del programa.



Cada una de las veces que el cuerpo del ciclo es ejecutado se llama **iteración**.

La condición es evaluada antes de cada iteración. Si la condición es inicialmente falsa, el ciclo no se ejecutará ninguna vez.

La sintaxis es la siguiente:

```
while condicion:
    sentencias
```

Por ejemplo, el siguiente programa multiplica dos números enteros sin usar el operador *:

```
m = int(raw_input())
n = int(raw_input())
p = 0
while m > 0:
    m = m - 1
    p = p + n
print 'El producto de m y n es', p
```

Para ver cómo funciona este programa, hagamos un ruteo con la entrada `m` = 4 y `n` = 7. La tabla 8.2 muestra cómo cambian los valores de todas las variables a medida que las sentencias van siendo ejecutadas.

En cada iteración, el valor de `m` decrece en 1. Cuando llega a 0, la condición del **`while`** deja de ser verdadera por lo que el ciclo termina. De este modo, se consigue que el resultado sea sumar `m` veces el valor de `n`.

Note que el ciclo no termina apenas el valor de `m` pasa a ser cero. La condición es evaluada una vez que la iteración completa ha terminado.

En general, el ciclo **`while`** se utiliza cuando no es posible saber de antemano cuántas veces será ejecutado el ciclo, pero sí qué es lo que tiene que ocurrir para que se termine.

8.5 Ciclo for con rango

El ciclo **for con rango** ejecuta una secuencia de sentencias una cantidad fija de veces.

Para llevar la cuenta, utiliza una **variable de control** que toma valores distintos en cada iteración.

Una de las sintaxis para usar un **`for`** con rango es la siguiente:

```
for variable in range(fin):
    que hacer para cada valor de la variable de control
```

En la primera iteración, la variable de control toma el valor 0. Al final de cada iteración, el valor de la variable aumenta automáticamente. El ciclo termina justo antes que la variable tome el valor `fin`.

Por ejemplo, el siguiente programa muestra los cubos de los números del 0 al 20:

```
for i in range(21):
    print i, i ** 3
```

Un **rango** es una sucesión de números enteros equiespaciados. Incluyendo la presentada más arriba, hay tres maneras de definir un rango:

```
range(final)
range(inicial, final)
range(inicial, final, incremento)
```

El valor inicial siempre es parte del rango. El valor final nunca es incluido en el rango. El incremento indica la diferencia entre dos valores consecutivos del rango.

Si el valor inicial es omitido, se supone que es 0. Si el incremento es omitido, se supone que es 1.

Con los ejemplos de la tabla 8.3 quedará más claro.

Usando un incremento negativo, es posible hacer ciclos que van hacia atrás:

```
for i in range(10, 0, -1):
    print i
print 'Feliz anno nuevo!'
```

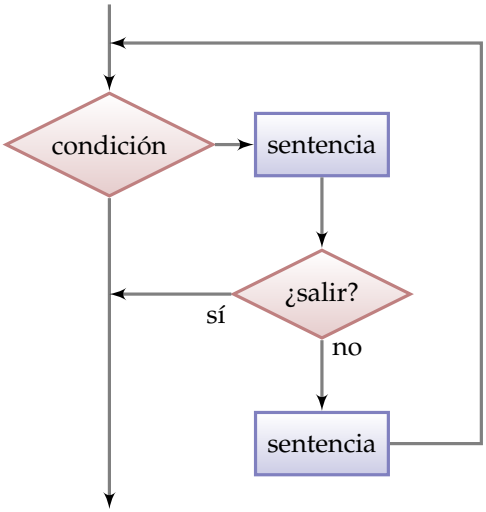
Rango	Valores que incluye
`range(9)`	0, 1, 2, 3, 4, 5, 6, 7, 8
`range(3, 13)`	3, 4, 5, 6, 7, 8, 9, 10, 11, 12
`range(3, 13, 2)`	3, 5, 7, 9, 11
`range(11, 4)`	Ningún valor
`range(11, 4, -1)`	11, 10, 9, 8, 7, 6, 5

Tabla 8.3: Ejemplos de rangos.

En general, el ciclo **`for`** con rango se usa cuando el número de iteraciones es conocido antes de entrar al ciclo.

8.6 Salir de un ciclo

Además de las condiciones de término propias de los ciclos **`while`** y **`for`**, siempre es posible salir de un ciclo en medio de una iteración usando la sentencia **`break`**:



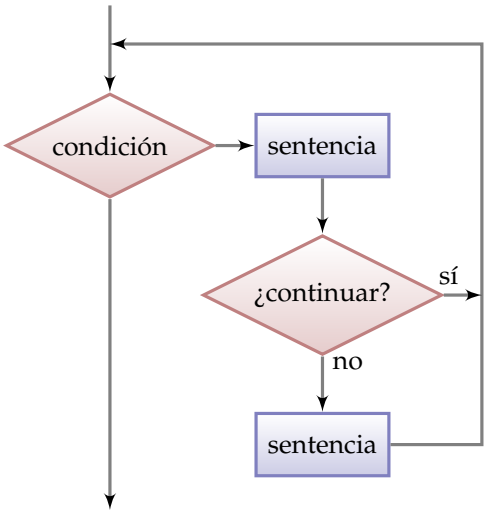
Por ejemplo, en el programa para determinar si un número es primo o no, la búsqueda de divisores puede ser terminada prematuramente apenas se encuentra el primero de ellos:

```
es_primo = True
for d in range(2, n):
    if n % d == 0:
        es_primo = False
        break
```

Es lógico es que el **break** aparezca siempre dentro de un **if**, pues de otro modo el ciclo terminaría siempre en la primera iteración.

8.7 Saltar a la siguiente iteración

La sentencia **continue** se usa para saltar a la iteración siguiente sin llegar al final de la que está en curso.



Por ejemplo, el siguiente programa muestra el seno, el coseno y la tangente de los números del 1 al 30, pero omitiendo los que terminan en 7:

```
from math import sin, cos, tan
for i in range(1, 31):
    if i % 10 == 7:
        continue
    print i, sin(i), cos(i), tan(i)
```

Capítulo 9

Patrones comunes

Como hemos visto hasta ahora, los programas son una combinación de asignaciones, condicionales y ciclos, organizados de tal manera que describan el algoritmo que queremos ejecutar.

Existen algunas tareas muy comunes y que casi siempre se resuelven de la misma manera. Por lo tanto, es conveniente conocerlas.

En programación, se llama **patrón** a una solución que es aplicable a un problema que ocurre a menudo. A continuación veremos algunos patrones comunes que ocurren en programación.

9.1 Sumar y multiplicar cosas

La suma y la multiplicación son operaciones binarias: operan sobre dos valores.

Para sumar y multiplicar más valores, generalmente dentro de un ciclo que los vaya generando, hay que usar una variable para ir guardando el resultado parcial de la operación. Esta variable se llama **acumulador**.

En el caso de la suma, el acumulador debe partir con el valor cero. Para la multiplicación, con el valor uno. En general, el acumulador debe ser inicializado con el elemento neutro de la operación que será aplicada.

Por ejemplo, el siguiente programa entrega el producto de los mil primeros números naturales:

```
producto = 1
for i in range(1, 1001):
    producto = producto * i

print producto
```

El siguiente programa entrega la suma de los cubos de los números naturales cuyo cuadrado es menor que mil:

```
i = 1
suma = 0
```

```
while i ** 2 < 1000:
    valor = i ** 3
    i = i + 1
    suma = suma + valor

print suma
```

En todos los casos, el patrón a seguir es algo como esto:

```
acumulador = valor_inicial
ciclo:
    valor = ...
    ...
    acumulador = acumulador operacion valor
```

El cómo adaptar esta plantilla a cada situación de modo que entregue el resultado correcto es responsabilidad del programador.

9.2 Contar cosas

Para contar cuántas veces ocurre algo, hay que usar un acumulador, al que se le suele llamar **contador**.

Tal como en el caso de la suma, debe ser inicializado en cero, y cada vez que aparezca lo que queremos contar, hay que incrementarlo en uno.

Por ejemplo, el siguiente programa cuenta cuántos de los números naturales menores que mil tienen un cubo terminado en siete:

```
c = 0
for i in range(1000):
    ultimo_digito = (i ** 3) % 10
    if ultimo_digito == 7:
        c = c + 1

print c
```

9.3 Encontrar el mínimo y el máximo

Para encontrar el máximo de una secuencia de valores, hay que usar un acumulador para recordar cuál es el mayor valor visto hasta el momento. En cada iteración, hay que examinar cuál es el valor actual, y si es mayor que el máximo, actualizar el acumulador.

El acumulador debe ser inicializado con un valor que sea menor a todos los valores que vayan a ser examinados.

Por ejemplo, el siguiente programa pide al usuario que ingrese diez números enteros positivos, e indica cuál es el mayor de ellos:

```
print 'Ingrese diez numeros positivos'
```

```
mayor = -1
for i in range(10):
    n = int(raw_input())
    if n > mayor:
        mayor = n

print 'El mayor es', mayor
```

Otra manera de hacerlo es reemplazando esta parte:

```
if n > mayor:
    mayor = n
```

por ésta:

```
mayor = max(mayor, n)
```

En este caso, como todos los números ingresados son positivos, inicializamos el acumulador en -1, que es menor que todos los valores posibles, por lo que el que sea el mayor eventualmente lo reemplazará.

¿Qué hacer cuando no exista un valor inicial que sea menor a todas las entradas posibles? Una solución tentativa es poner un número «muy negativo», y rezar para que el usuario no ingrese uno menor que él. Ésta no es la mejor solución, ya que no cubre todos los casos posibles:

```
mayor = -999999999
for i in range(10):
    n = int(raw_input())
    mayor = max(mayor, n)
```

Una opción más robusta es usar el primero de los valores por examinar:

```
mayor = int(raw_input())    # preguntar el primer valor
for i in range(9):          # preguntar los nueve siguientes
    n = int(raw_input())
    mayor = max(mayor, n)
```

La otra buena solución es usar explícitamente el valor $-\infty$, que en Python puede representarse usando el tipo `float` de la siguiente manera:

```
mayor = -float('inf')      # asi se dice "infinito" en Python
for i in range(10):
    n = int(raw_input())
    mayor = max(mayor, n)
```

Si sabemos de antemano que todos los números por revisar son positivos, podemos simplemente inicializar el acumulador en -1.

Por supuesto, para obtener el menor valor se hace de la misma manera, pero inicializando el acumulador con un número muy grande, y actualizándolo al encontrar un valor menor.

9.4 Generar pares

Para generar pares de cosas en un programa, es necesario usar dos ciclos anidados (es decir, uno dentro del otro). Ambos ciclos, el exterior y el interior, van asignando valores a sus variables de control, y ambas son accesibles desde dentro del doble ciclo.

Por ejemplo, todas las casillas de un tablero de ajedrez pueden ser identificadas mediante un par `(fila, columna)`. Para recorrer todas las casillas del tablero, se puede hacer de la siguiente manera:

```
for i in range(1, 9):
    for i in range(1, 9):
        print 'Casilla', i, j
```

Cuando los pares son desordenados (es decir, el par (a, b) es el mismo que el par (b, a)), el ciclo interior no debe partir desde cero, sino desde el valor que tiene la variable de control del ciclo interior.

Por ejemplo, el siguiente programa muestra todas las piezas de un juego de dominó:

```
for i in range(7):
    for j in range(i, 7):
        print i, j
```

Además, otros tipos de restricciones pueden ser necesarias. Por ejemplo, en un campeonato de fútbol, todos los equipos deben jugar entre ellos dos veces, una como local y una como visita. Por supuesto, no pueden jugar consigo mismos, por lo que es necesario excluir los pares compuestos por dos valores iguales. El siguiente programa muestra todos los partidos que se deben jugar en un campeonato con 6 equipos, suponiendo que los equipos están numerados del 0 al 5:

```
for i in range(6):
    for j in range(6):
        if i != j:
            print i, j
```

Otra manera de escribir el mismo código es:

```
for i in range(6):
    for j in range(6):
        if i == j:
            continue
        print i, j
```

Capítulo 10

Funciones

Supongamos que necesitamos escribir un programa que calcule el número combinatorio $C(m, n)$, definido como:

$$C(m, n) = \frac{m!}{(m-n)!\, n!},$$

donde $n!$ (el factorial de n) es el producto de los números enteros desde 1 hasta n:

$$n! = 1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n = \prod_{i=1}^{n} i.$$

El código para calcular el factorial de un número entero n es sencillo:

```
f = 1
for i in range(1, n + 1):
    f *= i
```

Sin embargo, para calcular el número combinatorio, hay que hacer lo mismo tres veces:

```
comb = 1

# multiplicar por m!
f = 1
for i in range(1, m + 1):
    f = f * i
comb = comb * f

# dividir por (m - n)!
f = 1
for i in range(1, m - n + 1):
    f = f * i
comb = comb / f
```

```
# dividir por n!
f = 1
for i in range(1, n + 1):
    f = f * i
comb = comb / f
```

La única diferencia entre los tres cálculos de factoriales es el valor de término de cada ciclo **for** (`m`, `m - n` y `n`, respectivamente).

Escribir el mismo código varias veces es tedioso y propenso a errores. Además, el código resultante es mucho más dificil de entender, pues no es evidente a simple vista qué es lo que hace.

Lo ideal sería que existiera una función llamada `factorial` que hiciera el trabajo sucio, y que pudiéramos usar de la siguiente manera:

```
factorial(m) / (factorial(m - n) * factorial(n))
```

Ya vimos anteriormente que Python ofrece «de fábrica» algunas funciones, como **int**, **min** y **abs**. Ahora veremos cómo crear nuestras propias funciones.

10.1 Funciones

En programación, una **función** es una sección de un programa que calcula un valor de manera independiente al resto del programa.

Una función tiene tres componentes importantes:

- los **parámetros**, que son los valores que recibe la función como entrada;
- el **código de la función**, que son las operaciones que hace la función; y
- el **resultado** (o **valor de retorno**), que es el valor final que entrega la función.

En esencia, una función es un mini programa. Sus tres componentes son análogos a la entrada, el proceso y la salida de un programa.

En el ejemplo del factorial, el parámetro es el entero al que queremos calcularle el factorial, el código es el ciclo que hace las multiplicaciones, y el resultado es el valor calculado.

10.2 Definición de funciones

Las funciones en Python son creadas con la sentencia **def**:

```
def nombre(parametros):
    # codigo de la funcion
```

Los parámetros son variables en las que quedan almacenados los valores de entrada.

La función contiene código igual al de cualquier programa. La diferencia es que, al terminar, debe entregar su resultado usando la sentencia **return**.

Por ejemplo, la función para calcular el factorial puede ser definida de la siguiente manera:

```
def factorial(n):
    f = 1
    for i in range(1, n + 1):
        f *= i
    return f
```

En este ejemplo, el resultado que entrega una llamada a la función es el valor que tiene la variable `f` al llegar a la última línea de la función.

Una vez creada, la función puede ser usada como cualquier otra, todas las veces que sea necesario:

```
>>> factorial(0)
1
>>> factorial(12) + factorial(10)
482630400
>>> factorial(factorial(3))
720
>>> n = 3
>>> factorial(n ** 2)
362880
```

Las variables que son creadas dentro de la función (incluyendo los parámetros y el resultado) se llaman **variables locales**, y sólo son visibles dentro de la función, no desde el resto del programa.

Por otra parte, las variables creadas fuera de alguna función se llaman **variables globales**, y son visibles desde cualquier parte del programa. Sin embargo, su valor no puede ser modificado, ya que una asignación crearía una variable local del mismo nombre.

En el ejemplo, las variables locales son `n`, `f` e `i`. Una vez que la llamada a la función termina, estas variables dejan de existir:

```
>>> factorial(5)
120
>>> f
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'f' is not defined
```

Después de definir la función `factorial`, podemos crear otra función llamada `comb` para calcular números combinatorios:

```
def comb(m, n):
    fact_m = factorial(m)
    fact_n = factorial(n)
    fact_m_n = factorial(m - n)
    c = fact_m / (fact_n * fact_m_n)
    return c
```

Esta función llama a `factorial` tres veces, y luego usa los resultados para calcular su resultado. La misma función puede ser escrita también de forma más sucinta:

```
def comb(m, n):
    return factorial(m) / (factorial(n) * factorial(m - n))
```

El programa completo es el siguiente:

```
def factorial(n):
    p = 1
    for i in range(1, n + 1):
        p *= i
    return p


def comb(m, n):
    return factorial(m) / (factorial(n) * factorial(m - n))


m = int(raw_input('Ingrese m: '))
n = int(raw_input('Ingrese n: '))
c = comb(m, n)
print '(m n) =', c
```

Note que, gracias al uso de las funciones, la parte principal del programa ahora tiene sólo cuatro líneas, y es mucho más fácil de entender.

10.3 Múltiples valores de retorno

En Python, una función puede retornar más de un valor. Por ejemplo, la siguiente función recibe una cantidad de segundos, y retorna el equivalente en horas, minutos y segundos:

```
def convertir_segundos(segundos):
    horas = segundos / (60 * 60)
    minutos = (segundos / 60) % 60
    segundos = segundos % 60
    return horas, minutos, segundos
```

Al llamar la función, se puede asignar un nombre a cada uno de los valores retornados:

```
>>> h, m, s = convertir_segundos(9814)
>>> h
2
>>> m
43
>>> s
34
```

Técnicamente, la función está retornando una **tupla** de valores, un tipo de datos que veremos más adelante:

```
>>> convertir_segundos(9814)
(2, 43, 34)
```

10.4 Funciones que no retornan nada

Una función puede realizar acciones sin entregar necesariamente un resultado. Por ejemplo, si un programa necesita imprimir cierta información muchas veces, conviene encapsular esta acción en una función que haga los **print**

```
def imprimir_datos(nombre, apellido, rol, dia, mes, anno):
    print 'Nombre completo:', nombre, apellido
    print 'Rol:', rol
    print 'Fecha de nacimiento:', dia, '/', mes, '/', anno

imprimir_datos('Perico', 'Los Palotes', '201101001-1',  3, 1, 1993)
imprimir_datos('Yayita', 'Vinagre',     '201101002-2', 10, 9, 1992)
imprimir_datos('Fulano', 'De Tal',      '201101003-3', 14, 5, 1990)
```

En este caso, cada llamada a la función `imprimir_datos` muestra los datos en la pantalla, pero no entrega un resultado. Este tipo de funciones son conocidas en programación como **procedimientos** o **subrutinas**, pero en Python son funciones como cualquier otra.

Técnicamente, todas las funciones retornan valores. En el caso de las funciones que no tienen una sentencia **return**, el valor de retorno siempre es **None**. Pero como la llamada a la función no aparece en una asignación, el valor se pierde, y no tiene ningún efecto en el programa.

Capítulo 11

Módulos

Un **módulo** (o **biblioteca**) es una colección de definiciones de variables, funciones y tipos (entre otras cosas) que pueden ser importadas para ser usadas desde cualquier programa.

Ya hemos visto algunos ejemplos de cómo usar módulos, particularmente el módulo matemático, del que podemos importar funciones como la exponencial y el coseno, y las constantes π y e:

```
>>> from math import exp, cos
>>> from math import pi, e
>>> print cos(pi / 3)
0.5
```

Las ventajas de usar módulos son:

- las funciones y variables deben ser definidas sólo una vez, y luego pueden ser utilizadas en muchos programas sin necesidad de reescribir el código;
- permiten que un programa pueda ser organizado en varias secciones lógicas, puestas cada una en un archivo separado;
- hacen más fácil compartir componentes con otros programadores.

Python viene «de fábrica» con muchos módulos listos para ser usados. Además, es posible descargar de internet e instalar módulos prácticamente para hacer cualquier cosa. Aquí aprenderemos a crear nuestros propios módulos.

11.1 Módulos provistos por Python

Éstos son algunos de los módulos estándares de Python, que pueden ser usados desde cualquier programa.

El módulo `math` contiene funciones y constantes matemáticas:

```
>>> from math import floor, radians
>>> floor(-5.9)
```

```
-6.0
>>> radians(180)
3.1415926535897931
```

El módulo `random` contiene funciones para producir números aleatorios (es decir, al azar):

```
>>> from random import choice, randrange, shuffle
>>> choice(['cara', 'sello'])
'cara'
>>> choice(['cara', 'sello'])
'sello'
>>> choice(['cara', 'sello'])
'sello'
>>> randrange(10)
7
>>> randrange(10)
2
>>> randrange(10)
5
>>> r = range(5)
>>> r
[0, 1, 2, 3, 4]
>>> shuffle(r)
>>> r
[4, 2, 0, 3, 1]
```

El módulo `datetime` provee tipos de datos para manipular fechas y horas:

```
>>> from datetime import date
>>> hoy = date(2011, 5, 31)
>>> fin_del_mundo = date(2012, 12, 21)
>>> (fin_del_mundo - hoy).days
570
```

El módulo `fractions` provee un tipo de datos para representar números racionales:

```
>>> from fractions import Fraction
>>> a = Fraction(5, 12)
>>> b = Fraction(9, 7)
>>> a + b
Fraction(143, 84)
```

El módulo `turtle` permite manejar una tortuga (¡haga la prueba!):

```
>>> from turtle import Turtle
>>> t = Turtle()
>>> t.forward(10)
>>> t.left(45)
```

```
>>> t.forward(20)
>>> t.left(45)
>>> t.forward(30)
>>> for i in range(10):
...   t.right(30)
...   t.forward(10 * i)
...
>>>
```

La lista completa de módulos de Python puede ser encontrada en la documentación de la biblioteca estándar.

11.2 Importación de nombres

La sentencia **import** importa objetos desde un módulo para poder ser usados en el programa actual.

Una manera de usar **import** es importar sólo los nombres específicos que uno desea utilizar en el programa:

```
from math import sin, cos
print sin(10)
print cos(20)
```

En este caso, las funciones `sin` y `cos` no fueron creadas por nosotros, sino importadas del módulo de matemáticas, donde están definidas.

La otra manera de usar **import** es importando el módulo completo, y accediendo a sus objetos mediante un punto:

```
import math
print math.sin(10)
print math.cos(10)
```

Las dos formas son equivalentes.

11.3 Creación de módulos

Un módulo sencillo es simplemente un archivo con código en Python. El nombre del archivo indica cuál es el nombre del módulo.

Por ejemplo, podemos crear un archivo llamado `pares.py` que tenga funciones relacionadas con los números pares:

```
def es_par(n):
    return n % 2 == 0

def es_impar(n):
    return not es_par(n)

def pares_hasta(n):
    return range(0, n, 2)
```

En este caso, el nombre del módulo es `pares`. Para poder usar estas funciones desde otro programa, el archivo `pares.py` debe estar en la misma carpeta que el programa.

Por ejemplo, un programa `mostrar_pares.py` puede ser escrito así:

```
from pares import pares_hasta

n = int(raw_input('Ingrese un entero: '))
print 'Los numeros pares hasta', n, 'son:'
for i in pares_hasta(n):
    print i
```

Y un programa `ver_si_es_par.py` puede ser escrito así:

```
import pares

n = int(raw_input('Ingrese un entero: '))
if pares.es_par(n):
    print n, 'es par'
else:
    print n, 'no es par'
```

Como se puede ver, ambos programas pueden usar los objetos definidos en el módulo simplemente importándolos.

11.4 Usar módulos como programas

Un archivo con extensión `.py` puede ser un módulo o un programa. Si es un módulo, contiene definiciones que pueden ser importadas desde un programa o desde otro módulo. Si es un programa, contiene código para ser ejecutado.

A veces, un programa también contiene definiciones (por ejemplo, funciones y variables) que también pueden ser útiles desde otro programa. Sin embargo, no pueden ser importadas, ya que al usar la sentencia **`import`** el programa completo sería ejecutado. Lo que ocurriría en este caso es que, al ejecutar el segundo programa, también se ejecutaría el primero.

Existe un truco para evitar este problema: siempre que hay código siendo ejecutado, existe una variable llamada `__name__`. Cuando se trata de un programa, el valor de esta variable es `'__main__'`, mientras que en un módulo, es el nombre del módulo.

Por lo tanto, se puede usar el valor de esta variable para marcar la parte del programa que debe ser ejecutada al ejecutar el archivo, pero no al importarlo.

Por ejemplo, el programa listado a continuación convierte unidades de medidas de longitud. Este programa es útil por sí solo, pero además sus cuatro funciones y las constantes `km_por_milla` y `cm_por_pulgada` podrían ser útiles en otro programa:

```
km_por_milla = 1.609344
cm_por_pulgada = 2.54
```

```
def millas_a_km(mi):
    return mi * km_por_milla

def km_a_millas(km):
    return km / km_por_milla

def pulgadas_a_cm(p):
    return p * cm_por_pulgada

def cm_a_pulgadas(cm):
    return cm / cm_por_pulgada

if __name__ == '__main__':
    print 'Que conversion desea hacer?'
    print '1) millas a kilometros'
    print '2) kilometros a millas'
    print '3) pulgadas a centimetros'
    print '4) centimetros a pulgadas'
    opcion = int(raw_input('> '))

    if opcion == 1:
        x = float(raw_input('Ingrese millas: '))
        print x, 'millas =', millas_a_km(x), 'km'
    elif opcion == 2:
        x = float(raw_input('Ingrese kilometros: '))
        print x, 'km =', km_a_millas(x), 'millas'
    elif opcion == 3:
        x = float(raw_input('Ingrese pulgadas: '))
        print x, 'in =', pulgadas_a_cm(x), 'cm'
    elif opcion == 4:
        x = float(raw_input('Ingrese centimetros: '))
        print x, 'cm =', cm_a_pulgadas(x), 'in'
    else:
        print 'Opcion incorrecta'
```

Al poner el cuerpo del programa dentro del `if __name__ == '__main__'`, el archivo puede ser usado como un módulo. Si no hiciéramos esto, cada vez que otro programa importe una función se ejecutaría el programa completo.

Haga la prueba: ejecute este programa, y luego escriba otro programa que importe alguna de las funciones. A continuación, haga lo mismo, pero eliminando el `if __name__ == '__main__'`.

Capítulo 12

Listas

Una **lista** es una colección ordenada de valores. Una lista puede contener cualquier cosa.

En Python, el tipo de datos que representa a las listas se llama `list`.

12.1 Cómo crear listas

Las dos maneras principales de crear una lista son:

- usar una lista literal, con los valores entre corchetes:

```
>>> primos = [2, 3, 5, 7, 11]
>>> primos
[2, 3, 5, 7, 11]
>>> []
[]
>>> [1.0 + 2.0, 3.0 + 4.0 + 5.0]
[3.0, 12.0]
>>> ['hola ' + 'mundo', 24 * 7, True or False]
['hola mundo', 168, True]
```

- usar la función `list` aplicada sobre un iterable:

```
>>> list('hola')
['h', 'o', 'l', 'a']
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list()
[]
```

12.2 Operaciones sobre listas

`len(l)` entrega el largo de la lista; es decir, cuántos elementos tiene:

```
>>> colores = ['azul', 'rojo', 'verde', 'amarillo']
>>> len(colores)
4
>>> len([True, True, True])
3
>>> len([])
0
```

`l[i]` entrega el `i`-ésimo valor de la lista. El valor `i` se llama **índice** del valor. Al igual que para los strings, los índices parten de cero:

```
>>> colores = ['azul', 'rojo', 'verde', 'amarillo']
>>> colores[0]
'azul'
>>> colores[3]
'amarillo'
```

Además, es posible modificar el valor del `i`-ésimo elemento:

```
>>> colores[1] = 'negro'
>>> colores
['azul', 'negro', 'verde', 'amarillo']
```

Si el índice `i` indica un elemento que no está en la lista, ocurre un **error de índice**:

```
>>> colores[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Si el índice es negativo, los elementos se cuentan desde el final hacia atrás:

```
>>> colores[-1]
'amarillo'
>>> colores[-4]
'azul'
```

`l.append(x)` agrega el elemento `x` al final de la lista:

```
>>> primos = [2, 3, 5, 7, 11]
>>> primos.append(13)
>>> primos.append(17)
>>> primos
[2, 3, 5, 7, 11, 13, 17]
```

Un comentario al margen: `append` es un **método**. Los métodos son funciones que están dentro de un objeto. Cada lista tiene su propia función `append`. Es importante tener esta distinción clara, ya que hay operaciones que están implementadas como funciones y otras como métodos.

`sum(x)` entrega la suma de los valores de la lista:

```
>>> sum([1, 2, 1, -1, -2])
1
>>> sum([])
0
```

`l1 + l2` concatena las listas `l1` y `l2`:

```
>>> list('perro') + [2, 3, 4]
['p', 'e', 'r', 'r', 'o', 2, 3, 4]
```

`l * n` repite `n` veces la lista `l`:

```
>>> [3.14, 6.28, 9.42] * 2
[3.14, 6.28, 9.42, 3.14, 6.28, 9.42]
>>> [3.14, 6.28, 9.42] * 0
[]
```

Para saber si un elemento `x` está en la lista `l`, se usa `x in l`. La versión negativa de **in** es **not in**:

```
>>> r = range(0, 20, 2)
>>> r
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> 12 in r
True
>>> 15 in r
False
>>> 15 not in r
True
```

`l[i:j]` es el operador de rebanado, que entrega una nueva lista que tiene desde el `i`-ésimo hasta justo antes del `j`-ésimo elemento de la lista `l`:

```
>>> x = [1.5, 3.3, 8.4, 3.1, 2.9]
>>> x[2:4]
[8.4, 3.1]
```

`l.count(x)` cuenta cuántas veces está el elemento `x` en la lista:

```
>>> letras = list('paralelepipedo')
>>> letras.count('p')
3
```

`l.index(x)` entrega cuál es el índice del valor `x`:

```
>>> colores = ['azul', 'rojo', 'verde', 'amarillo']
>>> colores.index('verde')
2
>>> colores.index('fucsia')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'fucsia' is not in list
```

`l.remove(x)` elimina el elemento `x` de la lista:

```
>>> l = [7, 0, 3, 9, 8, 2, 4]
>>> l.remove(2)
>>> l
[7, 0, 3, 9, 8, 4]
```

`del` `l[i]` elimina el `i`-ésimo elemento de la lista:

```
>>> l = [7, 0, 3, 9, 8, 2, 4]
>>> del l[2]
>>> l
[7, 0, 9, 8, 2, 4]
```

`l.reverse()` invierte la lista:

```
>>> l = [7, 0, 3, 9, 8, 2, 4]
>>> l.reverse()
>>> l
[4, 2, 8, 9, 3, 0, 7]
```

`l.sort()` ordena la lista:

```
>>> l = [7, 0, 3, 9, 8, 2, 4]
>>> l.sort()
>>> l
[0, 2, 3, 4, 7, 8, 9]
```

Para todas estas operaciones, siempre hay que tener muy claro si la lista es modificada o no. Por ejemplo, el rebanado no modifica la lista, sino que crea una nueva:

```
>>> ramos = ['Progra', 'Mate', 'Fisica', 'Ed.Fisica']
>>> ramos[:2]
['Progra', 'Mate']
>>> len(ramos)    # la lista sigue teniendo cuatro elementos
4
```

12.3 Iteración sobre una lista

Una lista es un objeto **iterable**. Esto significa que sus valores se pueden recorrer usando un ciclo **`for`**:

```
valores = [6, 1, 7, 8, 9]
for i in valores:
    print i ** 2
```

En cada iteración del **`for`**, la variable `i` toma uno de los valores de la lista, por lo que este programa imprime los valores 36, 1, 49, 64 y 81.

Capítulo 13

Tuplas

Una **tupla** es una secuencia de valores agrupados.

Una tupla sirve para agrupar, como si fueran un único valor, varios valores que, por su naturaleza, deben ir juntos.

El tipo de datos que representa a las tuplas se llama `tuple`. El tipo `tuple` es inmutable: una tupla no puede ser modificada una vez que ha sido creada.

Una tupla puede ser creada poniendo los valores separados por comas y entre paréntesis. Por ejemplo, podemos crear una tupla que tenga el nombre y el apellido de una persona:

```
>>> persona = ('Perico', 'Los Palotes')
>>> persona
('Perico', 'Los Palotes')
```

13.1 Desempaquetado de tuplas

Los valores individuales de una tupla pueden ser recuperados asignando la tupla a las variables respectivas. Esto se llama **desempaquetar la tupla** (en inglés: *unpack*):

```
>>> nombre, apellido = persona
>>> nombre
'Perico'
```

Si se intenta desempaquetar una cantidad incorrecta de valores, ocurre un error de valor:

```
>>> a, b, c = persona
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
```

Además, también es posible extraer los valores usando su índice, al igual que con las listas:

```
>>> persona[1]
'Los Palotes'
```

A diferencia de las listas, los elementos no se pueden modificar:

```
>>> persona[1] = 'Smith'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

13.2 Comparación de tuplas

Dos tuplas son iguales cuando tienen el mismo tamaño y cada uno de sus elementos correspondientes tienen el mismo valor:

```
>>> (1, 2) == (3 / 2, 1 + 1)
True
>>> (6, 1) == (6, 2)
False
>>> (6, 1) == (6, 1, 0)
False
```

Para determinar si una tupla es menor que otra, se utiliza lo que se denomina **orden lexicográfico**. Si los elementos en la primera posición de ambas tuplas son distintos, ellos determinan el ordenamiento de las tuplas:

```
>>> (1, 4, 7) < (2, 0, 0, 1)
True
>>> (1, 9, 10) < (0, 5)
False
```

La primera comparación es `True` porque `1 < 2`. La segunda comparación es `False` porque `1 > 0`. No importa el valor que tengan los siguientes valores, o si una tupla tiene más elementos que la otra.

Si los elementos en la primera posición son iguales, entonces se usa el valor siguiente para hacer la comparación:

```
>>> (6, 1, 8) < (6, 2, 8)
True
>>> (6, 1, 8) < (6, 0)
False
```

La primera comparación es `True` porque `6 == 6` y `1 < 2`. La segunda comparación es `False` porque `6 == 6` y `1 > 0`.

Si los elementos respectivos siguen siendo iguales, entonces se sigue probando con los siguientes uno por uno, hasta encontrar dos distintos. Si a una tupla se le acaban los elementos para comparar antes que a la otra, entonces es considerada menor que la otra:

```
>>> (1, 2) < (1, 2, 4)
True
>>> (1, 3) < (1, 2, 4)
False
```

La primera compación es `True` porque `1 == 1`, `2 == 2`, y ahí se acaban los elementos de la primera tupla. La segunda comparación es `False` porque `1 == 1` y `3 < 2`; en este caso sí se alcanza a determinar el resultado antes que se acaben los elementos de la primera tupla.

Este método de comparación es el mismo que se utiliza para poner palabras en orden alfabético (por ejemplo, en guías telefónicas y diccionarios):

```
>>> 'auto' < 'auxilio'
True
>>> 'auto' < 'autos'
True
>>> 'mes' < 'mesa' < 'mesadas' < 'mesas' < 'meses' < 'mi'
True
```

13.3 Usos típicos de las tuplas

Las tuplas se usan siempre que es necesario agrupar valores. Generalmente, conceptos del mundo real son representados como tuplas que agrupan información sobre ellos. Por ejemplo, un partido de fútbol se puede representar como una tupla de los equipos que lo juegan:

```
partido1 = ('Milan', 'Bayern')
```

Para representar puntos en el plano, se puede usar tuplas de dos elementos `(x, y)`. Por ejemplo, podemos crear una función `distancia` que recibe dos puntos y entrega la distancia entre ellos:

```
def distancia(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    dx = x2 - x1
    dy = y2 - y1
    return (dx ** 2 + dy ** 2) ** 0.5
```

Al llamar a la función, se le debe pasar dos tuplas:

```
>>> a = (2, 3)
>>> b = (7, 15)
>>> distancia(a, b)
13.0
```

Las fechas generalmente se representan como tuplas agrupando el año, el mes y el día. La ventaja de hacerlo en este orden (el año primero) es que las operaciones relacionales permiten saber en qué orden ocurrieron las fechas:

```
>>> hoy = (2011, 4, 19)
>>> ayer = (2011, 4, 18)
>>> navidad = (2011, 12, 25)
>>> anno_nuevo = (2012, 1, 1)
>>> hoy < ayer
False
>>> hoy < navidad < anno_nuevo
True
```

Una tupla puede contener otras tuplas. Por ejemplo, una persona puede ser descrita por su nombre, su rut y su fecha de nacimiento:

```
persona = ('Perico Los Palotes', '12345678-9', (1980, 5, 14))
```

En este caso, los datos se pueden desempaquetar así:

```
>>> nombre, rut, (a, m, d) = persona
>>> m
5
```

A veces a uno le interesan sólo algunos de los valores de la tupla. Para evitar crear variables innecesarias, se suele asignar estos valores a la variable _. Por ejemplo, si sólo nos interesa el mes en que nació la persona, podemos obtenerlo así:

```
>>> _, _, (_, mes, _) = persona
>>> mes
5
```

Una tabla de datos generalmente se representa como una lista de tuplas. Por ejemplo, la información de los alumnos que están tomando un ramo puede ser representada así:

```
alumnos = [
    ('Perico', 'Los Palotes', '201199001-5', 'Civil'),
    ('Fulano', 'De Tal',      '201199002-6', 'Electrica'),
    ('Fulano', 'De Tal',      '201199003-7', 'Mecanica'),
]
```

En este caso, se puede desempaquetar los valores automáticamente al recorrer la lista en un ciclo **for**:

```
for nombre, apellido, rol, carrera in alumnos:
    print nombre, 'estudia', carrera
```

O, ya que el apellido y el rol no son usados:

```
for nombre, _, _, carrera in alumnos:
    print nombre, 'estudia', carrera
```

Es posible crear tuplas de largo uno dejando una coma a continuación del único valor:

```
>>> t = (12,)
>>> len(t)
1
```

En otros lenguajes, las tuplas reciben el nombre de **registros**. Este nombre es común, por lo que conviene conocerlo.

13.4 Iteración sobre tuplas

Al igual que las listas, las tuplas son iterables:

```
for valor in (6, 1):
    print valor ** 2
```

Además, se puede convertir una tupla en una lista usando la función `list`, y una lista en una tupla usando la función `tuple`:

```
>>> a = (1, 2, 3)
>>> b = [4, 5, 6]
>>> list(a)
[1, 2, 3]
>>> tuple(b)
(4, 5, 6)
```

Capítulo 14

Diccionarios

Un **diccionario** es un tipo de datos que sirve para asociar pares de objetos.

Un diccionario puede ser visto como una colección de **llaves**, cada una de las cuales tiene asociada un **valor**. Las llaves no están ordenadas y no hay llaves repetidas. La única manera de acceder a un valor es a través de su llave.

14.1 Cómo crear diccionarios

Los diccionarios literales se crean usando paréntesis de llave (`{` y `}`). La llave y el valor van separados por dos puntos:

```
>>> telefonos = {'Pepito': 5552437, 'Jaimito': 5551428,
...              'Yayita': 5550012}
```

En este ejemplo, las llaves son `'Pepito'`, `'Jaimito'` y `'Yayita'`, y los valores asociados a ellas son, respectivamente, `5552437`, `5551428` y `5550012`.

Un diccionario vacío puede ser creado usando `{}` o con la función `dict()`:

```
>>> d = {}
>>> d = dict()
```

14.2 Cómo usar un diccionario

El valor asociado a la llave `k` en el diccionario `d` se puede obtener mediante `d[k]`:

```
>>> telefonos['Pepito']
5552437
>>> telefonos['Jaimito']
5551428
```

A diferencia de los índices de las listas, las llaves de los diccionarios no necesitan ser números enteros.

Si la llave no está presente en el diccionario, ocurre un **error de llave** (`KeyError`):

```
>>> telefonos['Fulanito']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Fulanito'
```

Se puede agregar una llave nueva simplemente asignándole un valor:

```
>>> telefonos['Susanita'] = 4448139
>>> telefonos
{'Pepito': 5552437, 'Susanita': 4448139, 'Jaimito': 5551428,
'Yayita': 5550012}
```

Note que el orden en que quedan las llaves en el diccionario no es necesariamente el mismo orden en que fueron agregadas.

Si se asigna un valor a una llave que ya estaba en el diccionario, el valor anterior se sobreescribe. Recuerde que un diccionario no puede tener llaves repetidas:

```
>>> telefonos
{'Pepito': 5552437, 'Susanita': 4448139, 'Jaimito': 5551428,
'Yayita': 5550012}
>>> telefonos['Jaimito'] = 4448139
>>> telefonos
{'Pepito': 5552437, 'Susanita': 4448139, 'Jaimito': 4448139,
'Yayita': 5550012}
```

Los valores sí pueden estar repetidos. En el ejemplo anterior, Jaimito y Susanita tienen el mismo número.

Para borrar una llave, se puede usar la sentencia **`del`**:

```
>>> del telefonos['Yayita']
>>> telefonos
{'Pepito': 5552437, 'Susanita': 4448139, 'Jaimito': 4448139}
```

Los diccionarios son iterables. Al iterar sobre un diccionario en un ciclo **`for`**, se obtiene las llaves:

```
>>> for k in telefonos:
...     print k
...
Pepito
Susanita
Jaimito
```

Para iterar sobre las llaves, se usa `d.values()`:

```
>>> for v in telefonos.values():
...     print v
...
5552437
4448139
4448139
```

Para iterar sobre las llaves y los valores simultáneamente, se usa el método `d.items()`:

```
>>> for k, v in telefonos.items():
...     print 'El telefono de', k, 'es', v
...
El telefono de Pepito es 5552437
El telefono de Susanita es 4448139
El telefono de Jaimito es 4448139
```

También es posible crear listas de llaves o valores:

```
>>> list(telefonos)
['Pepito', 'Susanita', 'Jaimito']
>>> list(telefonos.values())
[5552437, 4448139, 4448139]
```

`len(d)` entrega cuántos pares llave-valor hay en el diccionario:

```
>>> numeros = {15: 'quince', 24: 'veinticuatro'}
>>> len(numeros)
2
>>> len({})
0
```

`k in d` permite saber si la llave `k` está en el diccionario `d`:

```
>>> patas = {'gato': 4, 'humano': 2, 'pulpo': 8,
...          'perro': 4, 'ciempies': 100}
>>> 'perro' in patas
True
>>> 'gusano' in patas
False
```

Para saber si una llave *no* está en el diccionario, se usa el operador **not in**:

```
>>> 'gusano' not in patas
True
```

14.3 Restricciones sobre las llaves

No se puede usar cualquier objeto como llave de un diccionario. Las llaves deben ser de un tipo de datos inmutable. Por ejemplo, no se puede usar listas:

```
>>> d = {[1, 2, 3]: 'hola'}
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Típicamente, las llaves de los diccionarios suelen ser números, tuplas y strings.

Capítulo 15

Conjuntos

Un **conjunto** es una colección desordenada de valores no repetidos. Los conjuntos de Python son análogos a los conjuntos matemáticos. El tipo de datos que representa a los conjuntos se llama `set`. El tipo `set` es mutable: una vez que se ha creado un conjunto, puede ser modificado.

15.1 Cómo crear conjuntos

Las dos maneras principales de crear un conjunto son:

- usar un conjunto literal, entre llaves:

```
>>> colores = {'azul', 'rojo', 'blanco', 'blanco'}
>>> colores
{'rojo', 'azul', 'blanco'}
```

Note que el conjunto no incluye elementos repetidos, y que los elementos no quedan en el mismo orden en que fueron agregados.

- usar la función `set` aplicada sobre un iterable:

```
>>> set('abracadabra')
{'a', 'r', 'b', 'c', 'd'}
>>> set(range(50, 2000, 400))
{1250, 50, 1650, 850, 450}
>>> set([(1, 2, 3), (4, 5), (6, 7, 8, 9)])
{(4, 5), (6, 7, 8, 9), (1, 2, 3)}
```

El conjunto vacío debe ser creado usando `set()`, ya que `{}` representa al diccionario vacío.

Los elementos de un conjunto deben ser inmutables. Por ejemplo, no es posible crear un conjunto de listas, pero sí un conjunto de tuplas:

```
>>> s = {[2, 4], [6, 1]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> s = {(2, 4), (6, 1)}
>>>
```

Como un conjunto no es ordenado, no tiene sentido intentar obtener un elemento usando un índice:

```
>>> s = {'a', 'b', 'c'}
>>> s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

Sin embargo, sí es posible iterar sobre un conjunto usando un ciclo **for**:

```
>>> for i in {'a', 'b', 'c'}:
...     print i
...
a
c
b
```

15.2 Operaciones sobre conjuntos

`len(s)` entrega el número de elementos del conjunto `s`:

```
>>> len({'azul', 'verde', 'rojo'})
3
>>> len(set('abracadabra'))
5
>>> len(set())
0
```

`x in s` permite saber si el elemento `x` está en el conjunto `s`:

```
>>> 3 in {2, 3, 4}
True
>>> 5 in {2, 3, 4}
False
```

`x not in s` permite saber si `x` no está en `s`:

```
>>> 10 not in {2, 3, 4}
True
```

`s.add(x)` agrega el elemento `x` al conjunto `s`:

```
>>> s = {6, 1, 5, 4, 3}
>>> s.add(-37)
>>> s
{1, 3, 4, 5, 6, -37}
>>> s.add(4)
>>> s
{1, 3, 4, 5, 6, -37}
```

`s.remove(x)` elimina el elemento `x` del conjunto `s`:

```
>>> s = {6, 1, 5, 4, 3}
>>> s.remove(1)
>>> s
{3, 4, 5, 6}
```

Si el elemento `x` no está en el conjunto, ocurre un **error de llave**:

```
>>> s.remove(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 10
```

`&` y `|` son, respectivamente, los operadores de intersección y unión:

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 4, 6, 8}
>>> a & b
{2, 4}
>>> a | b
{1, 2, 3, 4, 6, 8}
```

`s - t` entrega la diferencia entre `s` y `t`; es decir, los elementos de `s` que no están en `t`:

```
>>> a - b
{1, 3}
```

`s ^ t` entrega la diferencia simétrica entre `s` y `t`; es decir, los elementos que están en `s` o en `t`, pero no en ambos:

```
>>> a ^ b
{1, 3, 6, 8}
```

El operador < aplicado sobre conjuntos significa «es subconjunto de»:

```
>>> {1, 2} < {1, 2, 3}
True
>>> {1, 4} < {1, 2, 3}
False
```

`s <= t` también indica si `s` es subconjunto de `t`. La distinción ocurre cuando los conjuntos son iguales:

```
>>> {1, 2, 3} < {1, 2, 3}
False
>>> {1, 2, 3} <= {1, 2, 3}
True
```

Capítulo 16

Procesamiento de texto

Hasta ahora, hemos visto cómo los tipos de datos básicos (strings, enteros, reales, booleanos) y las estructuras de datos permiten representar y manipular información compleja y abstracta en un programa.

Sin embargo, en muchos casos la información no suele estar disponible ya organizada en estructuras de datos convenientes de usar, sino en documentos de texto.

Por ejemplo, las páginas webs son archivos de puro texto, que describen la estructura de un documento en un lenguaje llamado HTML. Usted puede ver el texto de una página web buscando una instrucción «Ver código fuente» (o algo parecido) en el navegador. A partir de este texto, el navegador extrae la información necesaria para reconstruir la página que finalmente usted ve.

Un texto siempre es un string, que puede ser tan largo y complejo como se desee. El procesamiento de texto consiste en manipular strings, ya sea para extraer información del string, para convertir un texto en otro, o para codificar información en un string.

En Python, el tipo `str` provee muchos métodos convenientes para hacer procesamiento de texto, además de las operaciones más simples que ya aprendimos (como `s + t`, `s[i]` y `s in t`).

16.1 Saltos de línea

Un string puede contener caracteres de **salto de línea**, que tienen el efecto equivalente al de presionar la tecla Enter. El caracter de salto de línea se representa con `\n`:

```
>>> a = 'piano\nviolin\noboe'
>>> print a
piano
violin
oboe
```

Los saltos de línea sólo son visibles al imprimir el string usando la sentencia **print**. Si uno quiere ver el valor del string en la consola, el salto de línea aparecerá representado como `\n`:

```
>>> a
'piano\nviolin\noboe'
>>> print a
piano
violin
oboe
```

Aunque dentro del string se representa como una secuencia de dos símbolos, el salto de línea es un único caracter:

```
>>> len('a\nb')
3
```

En general, hay varios caracteres especiales que se representan comenzando con una barra invertida (`\`) seguida de una letra. Experimente, y determine qué significan los caracteres especiales `\t` y `\b`:

```
print 'abcde\tefg\thi\tjklm'
print 'abcde\befg\bhi\bjklm'
```

Para incluir una barra invertida dentro de un string, hay que hacerlo con `\\`:

```
>>> print 'c:\\>'
c:\>
>>> print '\\o/  o\n |  /|\\n/ \\ / \\'
\o/  o
 |  /|\
/ \ / \
```

16.2 Reemplazar secciones del string

El método `s.replace(antes, despues)` busca en el string `s` todas las apariciones del texto `antes` y las reemplaza por `despues`:

```
>>> 'La mar astaba sarana'.replace('a', 'e')
'Le mer estebe serene'
>>> 'La mar astaba sarana'.replace('a', 'i')
'Li mir istibi sirini'
>>> 'La mar astaba sarana'.replace('a', 'o')
'Lo mor ostobo sorono'
```

Hay que tener siempre muy claro que esta operación no modifica el string, sino que retorna uno nuevo:

```
>>> orden = 'Quiero arroz con pollo'
>>> orden.replace('arroz', 'pure').replace('pollo', 'huevo')
'Quiero pure con huevo'
>>> orden
'Quiero arroz con pollo'
```

16.3 Separar y juntar strings

`s.split()` separa el strings en varios strings, usando los espacios en blanco como separador. El valor retornado es una lista de strings:

```
>>> oracion = 'El veloz  murcielago hindu'
>>> oracion.split()
['El', 'veloz', 'murcielago', 'hindu']
```

Además, es posible pasar un parámetro al método `split` que indica cuál será el separador a usar en vez de los espacios en blanco:

```
>>> s = 'Ana lavaba las sabanas'
>>> s.split()
['Ana', 'lavaba', 'las', 'sabanas']
>>> s.split('a')
['An', ' l', 'v', 'b', ' l', 's s', 'b', 'n', 's']
>>> s.split('l')
['Ana ', 'avaba ', 'as sabanas']
>>> s.split('aba')
['Ana lav', ' las s', 'nas']
```

Esto es muy útil para pedir al usuario que ingrese datos en un programa de una manera más conveniente, y no uno por uno. Por ejemplo, antes hacíamos programas que funcionaban así:

```
Ingrese a: 2.3
Ingrese b: 1.9
Ingrese c: 2.3
El triangulo es isoceles.
```

Ahora podemos hacerlos así:

```
Ingrese lados del triangulo: 2.3 1.9 2.3
El triangulo es isoceles.
```

En este caso, el código del programa podría ser:

```
entrada = raw_input('Ingrese lados del triangulo: ')
lados = entrada.split()
a = int(lados[0])
b = int(lados[1])
c = int(lados[2])
print 'El triangulo es', tipo_triangulo(a, b, c)
```

O usando la función `map`, más simplemente:

```
entrada = raw_input('Ingrese lados del triangulo: ')
a, b, c = map(int, entrada.split())
print 'El triangulo es', tipo_triangulo(a, b, c)
```

`s.join(lista_de_strings)` une todos los strings de la lista, usando al string `s` como «pegamento»:

```
>>> valores = map(str, range(10))
>>> pegamento = ' '
>>> pegamento.join(valores)
'0 1 2 3 4 5 6 7 8 9'
>>> ''.join(valores)
'0123456789'
>>> ','.join(valores)
'0,1,2,3,4,5,6,7,8,9'
>>> ' --> '.join(valores)
'0 --> 1 --> 2 --> 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> 9'
```

16.4 Mayúsculas y minúsculas

`s.isupper()` y `s.islower()` indican si el string está, respectivamente, en mayúsculas o minúsculas:

```
>>> s = 'hola'
>>> t = 'Hola'
>>> u = 'HOLA'
>>> s.isupper(), s.islower()
(False, True)
>>> t.isupper(), t.islower()
(False, False)
>>> u.isupper(), u.islower()
(True, False)
```

`s.upper()` y `s.lower()` entregan el string `s` convertido, respectivamente, a mayúsculas y minúsculas:

```
>>> t
'Hola'
>>> t.upper()
'HOLA'
>>> t.lower()
'hola'
```

`s.swapcase()` cambia las minúsculas a mayúsculas, respectivamente, a mayúsculas y minúsculas:

```
>>> t.swapcase()
'hOLA'
```

16.5 Revisar contenidos del string

`s.startswith(t)` y `s.endswith(t)` indican si el string `s` comienza y termina, respectivamente, con el string `t`:

```
>>> objeto = 'paraguas'
>>> objeto.startswith('para')
True
>>> objeto.endswith('aguas')
True
>>> objeto.endswith('x')
False
>>> objeto.endswith('guaguas')
False
```

Nuestro conocido operador **in** indica si un string está contenido dentro de otro:

```
>>> 'pollo' in 'repollos'
True
>>> 'pollo' in 'gallinero'
False
```

16.6 Alineación de strings

Los métodos `s.ljust(n)`,`s.rjust(n)` y `s.center(n)` rellenan el string con espacios para que su largo sea igual a `n`, de modo que el contenido quede alineado, respectivamente, a la izquierda, a la derecha y al centro:

```
>>> contenido.ljust(20)
'hola                '
>>> contenido.center(20)
'        hola        '
>>> contenido.rjust(20)
'                hola'
```

Estos métodos son útiles para imprimir tablas bien alineadas:

```
datos = [
    ('Pepito', (1991, 12, 5), 'Osorno', '***'),
    ('Yayita', (1990, 1, 31), 'Arica', '*'),
    ('Fulanito', (1992, 10, 29), 'Porvenir', '****'),
]

for n, (a, m, d), c, e in datos:
    print n.ljust(10),
    print str(a).rjust(4), str(m).rjust(2), str(d).rjust(2),
    print c.ljust(10), e.center(5)
```

Este programa imprime lo siguiente:

```
Pepito     1991 12  5 Osorno      ***
Yayita     1990  1 31 Arica        *
Fulanito   1992 10 29 Porvenir    ****
```

16.7 Interpolación de strings

El método `format` permite usar un string como una plantilla que se puede completar con distintos valores dependiendo de la situación.

Las posiciones en que se deben rellenar los valores se indican dentro del string usando un número entre paréntesis de llaves:

```
>>> s = 'Soy {0} y vivo en {1}'
```

Estas posiciones se llaman *campos*. En el ejemplo, el string `s` tiene dos campos, numerados del cero al uno.

Para llenar los campos, hay que llamar al método `format` pasándole los valores como parámetros:

```
>>> s.format('Perico', 'Valparaiso')
'Soy Perico y vivo en Valparaiso'
>>> s.format('Erika', 'Berlin')
'Soy Erika y vivo en Berlin'
>>> s.format('Wang Dawei', 'Beijing')
'Soy Wang Dawei y vivo en Beijing'
```

El número indica en qué posición va el parámetro que está asociado al campo:

```
>>> '{1}{0}{2}{0}'.format('a', 'v', 'c')
'vaca'
>>> '{0} y {1}'.format('carne', 'huevos')
'carne y huevos'
>>> '{1} y {0}'.format('carne', 'huevos')
'huevos y carne'
```

Otra opción es referirse a los campos con un nombre. En este caso, hay que llamar al método `format` diciendo explícitamente el nombre del parámetro para asociarlo al valor:

```
>>> s = '{nombre} estudia en la {universidad}'
>>> s.format(nombre='Perico', universidad='UTFSM')
'Perico estudia en la UTFSM'
>>> s.format(nombre='Fulana', universidad='PUCV')
'Fulana estudia en la PUCV'
>>> s.format(universidad='UPLA', nombre='Yayita')
'Yayita estudia en la UPLA'
```

Capítulo 17

Archivos

Todos los datos que un programa utiliza durante su ejecución se encuentran en sus variables, que están almacenadas en la memoria RAM del computador.

La memoria RAM es un medio de almacenamiento **volátil**: cuando el programa termina, o cuando el computador se apaga, todos los datos se pierden para siempre.

Para que un programa pueda guardar datos de manera permanente, es necesario utilizar un medio de almacenamiento **persistente**, de los cuales el más importante es el disco duro.

Los datos en el disco duro están organizados en archivos. Un **archivo** es una secuencia de datos almacenados en un medio persistente que están disponibles para ser utilizados por un programa. Todos los archivos tienen un nombre y una ubicación dentro del sistema de archivos del sistema operativo.

Los datos en un archivo siguen estando presentes después de que termina el programa que lo ha creado. Un programa puede guardar sus datos en archivos para usarlos en una ejecución futura, e incluso puede leer datos desde archivos creados por otros programas.

Un programa no puede manipular los datos de un archivo directamente. Para usar un archivo, un programa siempre abrir el archivo y asignarlo a una variable, que llamaremos el **archivo lógico**. Todas las operaciones sobre un archivo se realizan a través del archivo lógico.

Dependiendo del contenido, hay muchos tipos de archivos. Nosotros nos preocuparemos sólo de los **archivos de texto**, que son los que contienen texto, y pueden ser abiertos y modificados usando un editor de texto como el Bloc de Notas. Los archivos de texto generalmente tienen un nombre terminado en `.txt`.

17.1 Lectura de archivos

Para leer datos de un archivo, hay que abrirlo de la siguiente manera:

```
archivo = open(nombre)
```

`nombre` es un string que tiene el nombre del archivo. `archivo` es el archivo lógico a través del que se manipulará el archivo.

Si el archivo no existe, ocurrirá un **error de entrada y salida** (`IOError`).

Es importante recordar que la variable `archivo` es una representación abstracta del archivo, y no los contenidos del mismo.

La manera más simple de leer el contenido es hacerlo línea por línea. Para esto, basta con poner el archivo lógico en un ciclo for:

```
for linea in archivo:
    # hacer algo
```

Una vez que los datos han sido leídos del archivo, hay que cerrarlo:

```
archivo.close()
```

Por ejemplo, supongamos que tenemos el archivo `himno.txt` que tiene el siguiente contenido:

```
Puro Chile
es tu cielo azulado
puras brisas
te cruzan tambien.
```

El archivo tiene cuatro líneas. Cada línea termina con un salto de línea (`\n`), que indica que a continuación comienza una línea nueva.

El siguiente programa imprime la primera letra de cada línea del himno:

```
archivo = open('himno.txt')
for linea in archivo:
    print linea[0]
archivo.close()
```

El ciclo for es ejecutado cuatro veces, una por cada línea del archivo. La salida del programa es:

```
P
e
p
t
```

Otro ejemplo: el siguiente programa imprime cuántos símbolos hay en cada línea:

```
archivo = open('himno.txt')
for linea in archivo:
    print len(linea)
archivo.close()
```

La salida es:

```
11
20
```

```
13
19
```

Note que el salto de línea (el "enter") es considerado en la cuenta:

P	u	r	o		C	h	i	l	e	\n

= 11 símbolos

Para obtener el string sin el salto de línea se puede usar el método `strip`, que elimina todos los símbolos de espaciado al principio y al final del string:

```
>>> s = '   Hola\n'
>>> s.strip()
'Hola'
```

Si modificamos el programa para eliminar el salto de línea:

```
archivo = open('himno.txt')
for linea in archivo:
    print len(linea.strip())
archivo.close()
```

entonces la salida es:

```
10
19
12
18
```

Lo importante es comprender que los archivos son leídos línea por línea usando el ciclo **`for`**.

17.2 Escritura en archivos

Los ejemplos anteriores suponen que el archivo por leer existe, y está listo para ser abierto y leído. Ahora veremos cómo crear los archivos y cómo escribir datos en ellos, para que otro programa después pueda abrirlos y leerlos.

Uno puede crear un archivo vacío abriéndolo de la siguiente manera:

```
archivo = open(nombre, 'w')
```

El segundo parámetro de la función `open` indica el uso que se le dará al archivo. `'w'` significa «escribir» (*write* en inglés).

Si el archivo señalado no existe, entonces será creado. Si ya existe, entonces será sobreescrito. Hay que tener cuidado entonces, pues esta operación elimina los datos del archivo que existía previamente.

Una vez abierto el archivo, uno puede escribir datos en él usando el método `write`:

```
a = open('prueba.txt', 'w')
a.write('Hola ')
a.write('mundo.')
a.close()
```

Una vez ejecutado este programa, el archivo `prueba.txt` será creado (o sobreescrito, si ya existía). Al abrirlo en el Bloc de Notas, veremos este contenido:

```
Hola mundo.
```

Para escribir varias líneas en el archivo, es necesario agregar explícitamente los saltos de línea en cada string que sea escrito. Por ejemplo, para crear el archivo `himno.txt` que usamos más arriba, podemos hacerlo así:

```
a = open('himno.txt', 'w')
a.write('Puro Chile\n')
a.write('es tu cielo azulado\n')
a.write('puras brisas\n')
a.write('te cruzan tambien.\n')
a.close()
```

Además del modo `'w'` (*write*), también existe el modo `'a'` (*append*), que permite escribir datos al final de un archivo existente. Por ejemplo, el siguiente programa abre el archivo `prueba.txt` que creamos más arriba, y agrega más texto al final de él:

```
a = open('prueba.txt', 'a')
a.write('\n')
a.write('Chao ')
a.write('pescao.')
a.close()
```

Si abrimos el archivo `prueba.txt` en el Bloc de Notas, veremos esto:

```
Hola mundo.

Chao pescao.
```

De haber abierto el archivo en modo `'w'` en vez de `'a'`, el contenido anterior (la frase `Hola mundo`) se habría borrado.

17.3 Archivos de valores con separadores

Una manera usual de almacenar datos con estructura de tabla en un archivo es la siguiente: cada línea del archivo representa una fila de la tabla, y los datos de una fila se ponen separados por algún símbolo especial.

Por ejemplo, supongamos que queremos guardar en un archivo los datos de esta tabla:

Nombre	Apellido	Nota 1	Nota 2	Nota 3	Nota 4
Perico	Los Palotes	90	75	38	65
Yayita	Vinagre	39	49	58	55
Fulana	De Tal	96	100	36	71

Si usamos el símbolo `:` como separador, el archivo `alumnos.txt` debería quedar así:

```
Perico:Los Palotes:90:75:38:65
Yayita:Vinagre:39:49:58:55
Fulanita:De Tal:96:100:36:71
```

El formato de estos archivos se suele llamar CSV, que en inglés son las siglas de *comma-separated values* (significa «valores separados por comas», aunque técnicamente el separador puede ser cualquier símbolo). A pesar del nombre especial que reciben, los archivos CSV son archivos de texto como cualquier otro, y se pueden tratar como tales.

Los archivos de valores con separadores son muy fáciles de leer y escribir, y por esto son muy usados. Como ejemplo práctico, si usted desea hacer un programa que analice los datos de una hoja de cálculo Excel, puede guardar el archivo con el formato CSV directamente en el Excel, y luego abrirlo desde su programa escrito en Python.

Para leer los datos de un archivo de valores con separadores, debe hacerlo línea por línea, eliminar el salto de línea usando el método `strip` y luego extraer los valores de la línea usando el método `split`. Por ejemplo, al leer la primera línea del archivo de más arriba obtendremos el siguiente string:

```
'Perico:Los Palotes:90:75:38:65\n'
```

Para separar los seis valores, lo podemos hacer así:

```
>>> linea.strip().split(':')
['Perico', 'Los Palotes', '90', '75', '38', '65']
```

Como se trata de un archivo de texto, todos los valores son strings. Una manera de convertir los valores a sus tipos apropiados es hacerlo uno por uno:

```
valores = linea.strip().split(':')
nombre   = valores[0]
apellido = valores[1]
nota1 = int(valores[2])
nota2 = int(valores[3])
nota3 = int(valores[4])
nota4 = int(valores[5])
```

Una manera más breve es usar las rebanadas y la función `map`:

```
valores = linea.strip().split(':')
nombre, apellido = valores[0:2]
nota1, nota2, nota3, nota4 = map(int, valores[2:6])
```

O podríamos dejar las notas en una lista, en vez de usar cuatro variables diferentes:

```
notas = map(int, valores[2:6])
```

Por ejemplo, un programa para imprimir el promedio de todos los alumnos se puede escribir así:

```
archivo_alumnos = open('alumnos.txt')
for linea in archivo_alumnos:
    valores = linea.strip().split(':')
    nombre, apellido = valores[0:2]
    notas = map(int, valores[2:6])
    promedio = sum(notas) / 4.0
    print '{0} obtuvo promedio {1}'.format(nombre, promedio)
archivo_alumnos.close()
```

Para escribir los datos en un archivo, hay que hacer el proceso inverso: convertir todos los datos al tipo string, pegarlos en un único string, agregar el salto de línea al final y escribir la línea en el archivo.

Si los datos de la línea ya están en una lista o una tupla, podemos convertirlos a string usando la función `map` y pegarlos usando el método `join`:

```
alumno = ('Perico', 'Los Palotes', 90, 75, 38, 65)
linea = ':'.join(map(str, alumno)) + '\n'
archivo.write(linea)
```

Otra manera es armar el string usando una plantilla:

```
linea = '{0}:{1}:{2}:{3}:{4}:{5}\n'.format(nombre, apellido,
                                           nota1, nota2,
                                           nota3, nota4)
archivo.write(linea)
```

Como siempre, usted debe preferir la manera que le parezca más simple de entender.

Capítulo 18

Arreglos

Las estructuras de datos que hemos visto hasta ahora (listas, tuplas, diccionarios, conjuntos) permiten manipular datos de manera muy flexible. Combinándolas y anidándolas, es posible organizar información de manera estructurada para representar sistemas del mundo real.

En muchas aplicaciones de ingeniería, por otra parte, más importante que la organización de los datos es la capacidad de hacer muchas operaciones a la vez sobre grandes conjuntos de datos numéricos de manera eficiente. Algunos ejemplos de problemas que requieren manipular grandes secuencias de números son: la predicción del clima, la construcción de edificios, y el análisis de indicadores financieros entre muchos otros.

La estructura de datos que sirve para almacenar estas grandes secuencias de números (generalmente de tipo `float`) es el **arreglo**.

Los arreglos tienen algunas similitudes con las listas:

- los elementos tienen un orden y se pueden acceder mediante su posición,
- los elementos se pueden recorrer usando un ciclo `for`.

Sin embargo, también tienen algunas restricciones:

- todos los elementos del arreglo deben tener el mismo tipo,
- en general, el tamaño del arreglo es fijo (no van creciendo dinámicamente como las listas),
- se usan principalmente para almacenar datos numéricos.

A la vez, los arreglos tienen muchas ventajas por sobre las listas, que iremos descubriendo a medida que avancemos en la materia.

Los arreglos son los equivalentes en programación de las **matrices** y **vectores** de las matemáticas. Precisamente, una gran motivación para usar arreglos es que hay mucha teoría detrás de ellos que puede ser usada en el diseño de algoritmos para resolver problemas interesantes.

18.1 Crear arreglos

El módulo que provee las estructuras de datos y las funciones para trabajar con arreglos se llama **NumPy**, y no viene incluído con Python, por lo que hay que instalarlo por separado.

Descargue el instalador apropiado para su versión de Python desde la página de descargas de NumPy. Para ver qué versión de Python tiene instalada, vea la primera línea que aparece al abrir una consola.

Para usar las funciones provistas por NumPy, debemos importarlas al principio del programa:

```
from numpy import array
```

Como estaremos usando frecuentemente muchas funciones de este módulo, conviene importarlas todas de una vez usando la siguiente sentencia:

```
from numpy import *
```

(Si no recuerda cómo usar el **import**, puede repasar el capítulo sobre módulos).

El tipo de datos de los arreglos se llama **array**. Para crear un arreglo nuevo, se puede usar la función **array** pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

```
>>> a = array([6, 1, 3, 9, 8])
>>> a
array([6, 1, 3, 9, 8])
```

Todos los elementos del arreglo tienen exactamente el mismo tipo. Para crear un arreglo de números reales, basta con que uno de los valores lo sea:

```
>>> b = array([6.0, 1, 3, 9, 8])
>>> b
array([ 6.,  1.,  3.,  9.,  8.])
```

Otra opción es convertir el arreglo a otro tipo usando el método `astype`:

```
>>> a
array([6, 1, 3, 9, 8])
>>> a.astype(float)
array([ 6.,  1.,  3.,  9.,  8.])
>>> a.astype(complex)
array([ 6.+0.j,  1.+0.j,  3.+0.j,  9.+0.j,  8.+0.j])
```

Hay muchas formas de arreglos que aparecen a menudo en la práctica, por lo que existen funciones especiales para crearlos:

- **zeros**(n) crea un arreglo de n ceros;
- **ones**(n) crea un arreglo de n unos;

- `arange(a, b, c)` crea un arreglo de forma similar a la función `range`, con las diferencias que `a`, `b` y `c` pueden ser reales, y que el resultado es un arreglo y no una lista;
- `linspace(a, b, n)` crea un arreglo de `n` valores equiespaciados entre `a` y `b`.

```
>>> zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])

>>> ones(5)
array([ 1.,  1.,  1.,  1.,  1.])

>>> arange(3.0, 9.0)
array([ 3.,  4.,  5.,  6.,  7.,  8.])

>>> linspace(1, 2, 5)
array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ])
```

18.2 Operaciones con arreglos

Las limitaciones que tienen los arreglos respecto de las listas son compensadas por la cantidad de operaciones convenientes que permiten realizar sobre ellos.

Las operaciones aritméticas entre arreglos se aplican elemento a elemento:

```
>>> a = array([55, 21, 19, 11,  9])
>>> b = array([12, -9,  0, 22, -9])

# sumar los dos arreglos elemento a elemento
>>> a + b
array([67, 12, 19, 33,  0])

# multiplicar elemento a elemento
>>> a * b
array([ 660, -189,    0,  242,  -81])

# restar elemento a elemento
>>> a - b
array([ 43,  30,  19, -11,  18])
```

Las operaciones entre un arreglo y un valor simple funcionan aplicando la operación a todos los elementos del arreglo, usando el valor simple como operando todas las veces:

```
>>> a
array([55, 21, 19, 11,  9])
```

```
# multiplicar por 0.1 todos los elementos
>>> 0.1 * a
array([ 5.5,  2.1,  1.9,  1.1,  0.9])

# restar 9.0 a todos los elementos
>>> a - 9.0
array([ 46.,  12.,  10.,   2.,   0.])
```

Note que si quisiéramos hacer estas operaciones usando listas, necesitaríamos usar un ciclo para hacer las operaciones elemento a elemento.

Las operaciones relacionales también se aplican elemento a elemento, y retornan un arreglo de valores booleanos:

```
>>> a = array([5.1, 2.4, 3.8, 3.9])
>>> b = array([4.2, 8.7, 3.9, 0.3])
>>> c = array([5, 2, 4, 4]) + array([1, 4, -2, -1]) / 10.0
>>> a < b
array([False,  True,  True, False], dtype=bool)
>>> a == c
array([ True,  True,  True,  True], dtype=bool)
```

Para reducir el arreglo de booleanos a un único valor, se puede usar las funciones `any` y `all`. `any` retorna **`True`** si al menos uno de los elementos es verdadero, mientras que `all` retorna **`True`** sólo si todos lo son (en inglés, *any* signfica «alguno», y *all* significa «todos»):

```
>>> any(a < b)
True
>>> any(a == b)
False
>>> all(a == c)
True
```

18.3 Funciones sobre arreglos

NumPy provee muchas funciones matemáticas que también operan elemento a elemento. Por ejemplo, podemos obtener el seno de 9 valores equiespaciados entre 0 y $\pi/2$ con una sola llamada a la función `sin`:

```
>>> from numpy import linspace, pi, sin

>>> x = linspace(0, pi/2, 9)
>>> x
array([ 0.        ,  0.19634954,  0.39269908,
        0.58904862,  0.78539816,  0.9817477 ,
        1.17809725,  1.37444679,  1.57079633])
```

```
>>> sin(x)
array([ 0.        ,  0.19509032,  0.38268343,
        0.55557023,  0.70710678,  0.83146961,
        0.92387953,  0.98078528,  1.        ])
```

Como puede ver, los valores obtenidos crecen desde 0 hasta 1, que es justamente como se comporta la función seno en el intervalo $[0, \pi/2]$.

Aquí también se hace evidente otra de las ventajas de los arreglos: al mostrarlos en la consola o al imprimirlos, los valores aparecen perfectamente alineados. Con las listas, esto no ocurre:

```
>>> list(sin(x))
[0.0, 0.19509032201612825, 0.38268343236508978, 0.5555702330
1960218, 0.70710678118654746, 0.83146961230254524, 0.9238795
3251128674, 0.98078528040323043, 1.0]
```

18.4 Arreglos aleatorios

El módulo NumPy contiene a su vez otros módulos que proveen funcionalidad adicional a los arreglos y funciones básicos.

El módulo `numpy.random` provee funciones para crear **números aleatorios** (es decir, generados al azar), de las cuales la más usada es la función `random`, que entrega un arreglo de números al azar distribuidos uniformemente entre 0 y 1:

```
>>> from numpy.random import random
>>> random(3)
array([ 0.53077263,  0.22039319,  0.81268786])
>>> random(3)
array([ 0.07405763,  0.04083838,  0.72962968])
>>> random(3)
array([ 0.51886706,  0.46220545,  0.95818726])
```

18.5 Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice -1:

```
>>> a = array([6.2, -2.3, 3.4, 4.7, 9.8])

>>> a[0]
6.2
>>> a[1]
-2.3
```

```
>>> a[-2]
4.7
>>> a[3]
4.7
```

Una seccion del arreglo puede ser obtenida usando el operador de rebanado `a[i:j]`. Los índices `i` y `j` indican el rango de valores que serán entregados:

```
>>> a
array([ 6.2, -2.3,  3.4,  4.7,  9.8])
>>> a[1:4]
array([-2.3,  3.4,  4.7])
>>> a[2:-2]
array([ 3.4])
```

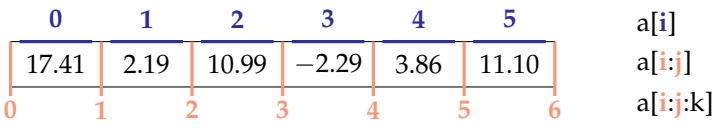
Si el primer índice es omitido, el rebanado comienza desde el principio del arreglo. Si el segundo índice es omitido, el rebanado termina al final del arreglo:

```
>>> a[:2]
array([ 6.2, -2.3])
>>> a[2:]
array([ 3.4,  4.7,  9.8])
```

Un tercer índice puede indicar cada cuántos elementos serán incluídos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.   ,  0.125,  0.25 ,  0.375,  0.5  ,  0.625,
0.75 ,  0.875,  1.   ])
>>> a[1:7:2]
array([ 0.125,  0.375,  0.625])
>>> a[::3]
array([ 0.   ,  0.375,  0.75 ])
>>> a[-2::-2]
array([ 0.875,  0.625,  0.375,  0.125])
>>> a[::-1]
array([ 1.   ,  0.875,  0.75 ,  0.625,  0.5  ,  0.375,
0.25 ,  0.125,  0.   ])
```

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:



```
>>> b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])
>>> b[2:5]
array([ 10.99,  -2.29,   3.86])
>>> b[:5]
array([ 17.41,   2.19,  10.99,  -2.29,   3.86])
>>> b[1:1]
array([], dtype=float64)
>>> b[1:5:2]
array([ 2.19, -2.29])
```

18.6 Algunos métodos convenientes

Los arreglos proveen algunos métodos útiles que conviene conocer.

Los métodos `min` y `max`, entregan respectivamente el mínimo y el máximo de los elementos del arreglo:

```
>>> a = array([4.1, 2.7, 8.4, pi, -2.5, 3, 5.2])
>>> a.min()
-2.5
>>> a.max()
8.4000000000000004
```

Los métodos `argmin` y `argmax` entregan respectivamente la posición del mínimo y del máximo:

```
>>> a.argmin()
4
>>> a.argmax()
2
```

Los métodos `sum` y `prod` entregan respectivamente la suma y el producto de los elementos:

```
>>> a.sum()
24.041592653589795
>>> a.prod()
-11393.086289208301
```

Capítulo 19

Arreglos bidimensionales

Los **arreglos bidimensionales** son tablas de valores. Cada elemento de un arreglo bidimensional está simultáneamente en una fila y en una columna.

En matemáticas, a los arreglos bidimensionales se les llama matrices, y son muy utilizados en problemas de Ingeniería.

En un arreglo bidimensional, cada elemento tiene una posición que se identifica mediante dos índices: el de su fila y el de su columna.

19.1 Crear arreglos bidimensionales

Los arreglos bidimensionales también son provistos por NumPy, por lo que debemos comenzar importando las funciones de este módulo:

```
from numpy import *
```

Al igual que los arreglos de una dimensión, los arreglos bidimensionales también pueden ser creados usando la función `array`, pero pasando como argumentos una lista con las filas de la matriz:

```
a = array([[5.1, 7.4, 3.2, 9.9],
           [1.9, 6.8, 4.1, 2.3],
           [2.9, 6.4, 4.3, 1.4]])
```

Todas las filas deben ser del mismo largo, o si no ocurre un error de valor:

```
>>> array([[1], [2, 3]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: setting an array element with a sequence.
```

Los arreglos tienen un atributo llamado `shape`, que es una tupla con los tamaños de cada dimensión. En el ejemplo, `a` es un arreglo de dos dimensiones que tiene tres filas y cuatro columnas:

```
>>> a.shape
(3, 4)
```

Los arreglos también tienen otro atributo llamado `size` que indica cuántos elementos tiene el arreglo:

```
>>> a.size
12
```

Por supuesto, el valor de `a.size` siempre es el producto de los elementos de `a.shape`.

Hay que tener cuidado con la función **`len`**, ya que no retorna el tamaño del arreglo, sino su cantidad de filas:

```
>>> len(a)
3
```

Las funciones **`zeros`** y **`ones`** también sirven para crear arreglos bidimensionales. En vez de pasarles como argumento un entero, hay que entregarles una tupla con las cantidades de filas y columnas que tendrá la matriz:

```
>>> zeros((3, 2))
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> ones((2, 5))
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

Lo mismo se cumple para muchas otras funciones que crean arreglos; por ejemplom la función `random`:

```
>>> from numpy.random import random
>>> random((5, 2))
array([[ 0.80177393,  0.46951148],
       [ 0.37728842,  0.72704627],
       [ 0.56237317,  0.3491332 ],
       [ 0.35710483,  0.44033758],
       [ 0.04107107,  0.47408363]])
```

19.2 Operaciones con arreglos bidimensionales

Al igual que los arreglos de una dimensión, las operaciones sobre las matrices se aplican término a término:

```
>>> a = array([[5, 1, 4],
...            [0, 3, 2]])
>>> b = array([[2, 3, -1],
...            [1, 0, 1]])
>>> a + 2
array([[7, 3, 6],
       [2, 5, 4]])
```

```
>>> a ** b
array([[25,  1,  0],
       [ 0,  1,  2]])
```

Cuando dos matrices aparecen en una operación, ambas deben tener exactamente la misma forma:

```
>>> a = array([[5, 1, 4],
...            [0, 3, 2]])
>>> b = array([[ 2,  3],
...            [-1,  1],
...            [ 0,  1]])
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be
            broadcast to a single shape
```

19.3 Obtener elementos de un arreglo bidimensional

Para obtener un elemento de un arreglo, debe indicarse los índices de su fila `i` y su columna `j` mediante la sintaxis `a[i, j]`:

```
>>> a = array([[ 3.21,  5.33,  4.67,  6.41],
               [ 9.54,  0.30,  2.14,  6.57],
               [ 5.62,  0.54,  0.71,  2.56],
               [ 8.19,  2.12,  6.28,  8.76],
               [ 8.72,  1.47,  0.77,  8.78]])
>>> a[1, 2]
2.14
>>> a[4, 3]
8.78
>>> a[-1, -1]
8.78
>>> a[0, -1]
6.41
```

También se puede obtener secciones rectangulares del arreglo usando el operador de rebanado con los índices:

```
>>> a[2:3, 1:4]
array([[ 0.54,  0.71,  2.56]])
>>> a[1:4, 0:4]
array([[ 9.54,  0.3 ,  2.14,  6.57],
       [ 5.62,  0.54,  0.71,  2.56],
       [ 8.19,  2.12,  6.28,  8.76]])
>>> a[1:3, 2]
array([ 2.14,  0.71])
```

```
>>> a[0:4:2, 3:0:-1]
array([[ 6.41,  4.67,  5.33],
       [ 2.56,  0.71,  0.54]])
>>> a[::4, ::3]
array([[ 3.21,  6.41],
       [ 8.72,  8.78]])
```

Para obtener una fila completa, hay que indicar el índice de la fila, y poner `:` en el de las columnas (significa «desde el principio hasta el final»). Lo mismo para las columnas:

```
>>> a[2, :]
array([ 5.62,  0.54,  0.71,  2.56])
>>> a[:, 3]
array([ 6.41,  6.57,  2.56,  8.76,  8.78])
```

Note que el número de dimensiones es igual a la cantidad de rebanados que hay en los índices:

```
>>> a[2, 3]      # valor escalar (cero dimensiones)
2.56
>>> a[2:3, 3]    # arreglo de una dimension de 1 elemento
array([ 2.56])
>>> a[2:3, 3:4]  # arreglo de dos dimensiones de 1 x 1
array([[ 2.56]])
```

19.4 Otras operaciones

La **trasposicion** consiste en cambiar las filas por las columnas y viceversa. Para trasponer un arreglo, se usa el método `transpose`:

```
>>> a
array([[ 3.21,  5.33,  4.67,  6.41],
       [ 9.54,  0.3 ,  2.14,  6.57],
       [ 5.62,  0.54,  0.71,  2.56]])
>>> a.transpose()
array([[ 3.21,  9.54,  5.62],
       [ 5.33,  0.3 ,  0.54],
       [ 4.67,  2.14,  0.71],
       [ 6.41,  6.57,  2.56]])
```

El método `reshape` entrega un arreglo que tiene los mismos elementos pero otra forma. El parámetro de `reshape` es una tupla indicando la nueva forma del arreglo:

```
>>> a = arange(12)
>>> a
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
>>> a.reshape((4, 3))
```

```
array([[ 0, 1, 2],
       [ 3, 4, 5],
       [ 6, 7, 8],
       [ 9, 10, 11]])
>>> a.reshape((2, 6))
array([[ 0, 1, 2, 3, 4, 5],
       [ 6, 7, 8, 9, 10, 11]])
```

La función `diag` aplicada a un arreglo bidimensional entrega la diagonal principal de la matriz (es decir, todos los elementos de la forma `a[i, i]`):

```
>>> a
array([[ 3.21,  5.33,  4.67,  6.41],
       [ 9.54,  0.3 ,  2.14,  6.57],
       [ 5.62,  0.54,  0.71,  2.56]])
>>> diag(a)
array([ 3.21,  0.3 ,  0.71])
```

Además, `diag` recibe un segundo parámetro opcional para indicar otra diagonal que se desee obtener. Las diagonales sobre la principal son positivas, y las que están bajo son negativas:

```
>>> diag(a, 2)
array([ 4.67,  6.57])
>>> diag(a, -1)
array([ 9.54,  0.54])
```

La misma función `diag` también cumple el rol inverso: al recibir un arreglo de una dimensión, retorna un arreglo bidimensional que tiene los elementos del parámetro en la diagonal:

```
>>> diag(arange(5))
array([[0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 4]])
```

Capítulo 20

Interfaces gráficas

Todo programa necesita contar con algún mecanismo para recibir la entrada y entregar la salida. Ya hemos visto dos maneras de hacer entrada:

- entrada por teclado (`raw_input`), y
- entrada por archivo (`for linea in archivo: ...`);

y dos maneras de hacer salida:

- salida por consola (`print`), y
- salida por archivo (`archivo.write(...)`).

La mayoría de los programas que usamos a diario no funcionan así, sino que tienen una **interfaz gráfica**, compuesta por ventanas, menúes, botones y otros elementos, a través de los cuales podemos interactuar con el programa.

Los programas con interfaces gráficas son fundamentalmente diferentes a los programas con interfaces de texto. Los programas que hemos escrito hasta ahora se ejecutan completamente de principio a fin, deteniéndose sólo cuando debemos ingresar datos.

Los programas gráficos, por otra parte, realizan acciones sólo cuando ocurren ciertos eventos provocados por el usuario (como hacer clic en un botón, o escribir algo en una casilla de texto), y el resto del tiempo se quedan esperando que algo ocurra, sin hacer nada. El programa no tiene control sobre cuándo debe hacer algo. Esto requiere que los programas sean estructurados de una manera especial, que iremos aprendiendo de a poco.

Python incluye un módulo llamado `Tkinter` que provee todas las funciones necesarias, que deben ser importadas al principio del programa:

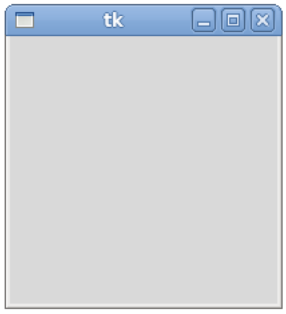
```
from Tkinter import *
```

20.1 Creación de la ventana

El siguiente programa es la interfaz gráfica más simple que se puede crear:

```
from Tkinter import *
w = Tk()
w.mainloop()
```

Haga la prueba: copie este programa en el editor de texto, guárdelo y ejecútelo. Debería aparecer una ventana vacía:



La sentencia `w = Tk()` crea la ventana principal del programa, y la asigna a la variable `w`. Toda interfaz gráfica debe tener una ventana principal en la que se irán agregando cosas. Esta línea va al principio del programa.

La sentencia `w.mainloop()` indica a la interfaz que debe quedarse esperando a que el usuario haga algo. Esta línea siempre debe ir al final del programa.

Al ejecutarlo, puede darse cuenta que el programa no termina. Esto ocurre porque la llamada al método `mainloop()` se «queda pegada» esperando que algo ocurra. Esto se llama un **ciclo de eventos**, y es simplemente un ciclo infinito que está continuamente esperando que algo ocurra.

Todos los programas con interfaz gráfica deben seguir esta estructura: la creación de la ventana al principio del programa y la llamada al ciclo de eventos al final del programa.

20.2 Creación de widgets

Un **widget** es cualquier cosa que uno puede poner en una ventana. Por ahora, veremos tres tipos de widgets sencillos, que son suficientes para crear una interfaz gráfica funcional:

- las **etiquetas** (`Label`) sirven para mostrar datos,
- los **botones** (`Button`) sirven para hacer que algo ocurra en el programa, y
- los **campos de entrada** (`Entry`) sirven para ingresar datos al programa.

En un programa en ejecución, estos widgets se ven así:

El `Entry` es análogo al **`raw_input`** de los programas de consola: sirve para que el programa reciba la entrada. El `Label` es análogo al **`print`**: sirve para que el programa entregue la salida.

Un botón puede ser visto como un «llamador de funciones»: cada vez que un botón es presionado, se hace una llamada a la función asociada a ese botón. Los botones no tienen un análogo, pues los programas de consola se ejecutan de principio a fin inmediatamente, y por esto no necesitan que las llamadas a las funciones sean gatilladas por el usuario.

Para agregar un widget a un programa, hay que ocupar las funciones con los nombres de los widgets (`Label`, `Button` y `Entry`). Estas funciones reciben como primer parámetro obligatorio la ventana que contendrá el widget. Además, tienen parámetros opcionales que deben ser pasados usando la sintaxis de asignación de parámetros por nombre. Por ejemplo, el parámetro `text` sirve para indicar cuál es el texto que aparecerá en un botón o en una etiqueta.

Por ejemplo, la siguiente sentencia crea un botón con el texto `Saludar`, contenido en la ventana `w`:

```
b = Button(w, text='Saludar')
```

Si bien esto crea el botón y lo asigna a la variable `b`, el botón no es agregado a la ventana `w` inmediatamente: lo que hicimos fue simplemente decirle al botón cuál es su contenedor, para que lo tenga en cuenta al momento de ser agregado. Para que esto ocurra, debemos llamar al método `pack`, que es una manera de decirle al widget «empaquétate dentro de tu contenedor»:

```
b.pack()
```

Como referencia, el programa que crea la ventana de la imagen es el siguiente (¡pruébelo!):

```
from Tkinter import *

w = Tk()

l = Label(w, text='Etiqueta')
l.pack()

b = Button(w, text='Boton')
b.pack()
```

```
e = Entry(w)
e.pack()

w.mainloop()
```

Los widgets van siendo apilados verticalmente, desde arriba hacia abajo, en el mismo orden en que van siendo apilados. Ya veremos cómo empaquetarlos en otras direcciones.

20.3 Controladores

Al crear un botón de la siguiente manera:

```
b = Button(w, text='Saludar')
```

no hay ninguna acción asociada a él. Al hacer clic en el botón, nada ocurrirá.

Para que ocurra algo al hacer clic en el botón, hay que asociarle una acción. Un **controlador** es una función que será ejecutada al hacer clic en un botón.

Los controladores deben ser funciones que no reciben ningún parámetro.

Por ejemplo, supongamos que queremos que el programa imprima el mensaje `Hola` en la consola cada vez que se haga clic en el botón que dice «Saludar». Primero, hay que crear el controlador:

```
def saludar():
    print 'Hola'
```

Para asociar el controlador al botón, hay que pasarlo a través del parámetro `command` (en inglés: «orden») al momento de crear el botón:

```
b = Button(w, text='Saludar', command=saludar)
```

Esta línea significa: crear el botón `b`, contenido en la ventana `w`, que tenga el texto `'Saludar'` y que al hacer clic en él se ejecute la función `saludar`.

El siguiente ejemplo es un programa completo que tiene dos botones: uno para saludar y otro para salir del programa. El controlador del segundo botón es la función `exit`, que ya viene con Python:

```
from Tkinter import *

def saludar():
    print 'Hola'

w = Tk()

l = Label(w, text='Hola progra')
l.pack()

b1 = Button(w, text='Saludar', command=saludar)
b1.pack()
```

```
b2 = Button(w, text='Salir', command=exit)
b2.pack()

w.mainloop()
```

El programa se ve así:

Ejecute el programa, y pruebe lo que ocurre al hacer clic en ambos botones.

20.4 Modelos

Mediante el uso de controladores, ya podemos hacer interfaces que hagan algo, pero que siguen teniendo una limitación: las interfaces sólo reaccionan a eventos que ocurren, pero no tienen memoria para recordar información.

Un **modelo** es un dato almacenado que está asociado a la interfaz. Usando modelos, se puede lograr que la interfaz vaya cambiando su estado interno a medida que ocurren eventos.

En general, a la hora de crear un programa con interfaz gráfica, debemos crear un modelo para cada dato que deba ser recordado durante el programa.

Tkinter provee varios tipos de modelos, pero para simplificar podemos limitarnos a usar sólo modelos de tipo string. Un modelo puede ser creado de la siguiente manera:

```
m = StringVar()
```

Aquí, el modelo `m` es capaz de recordar un string

Para modificar el valor del modelo `m`, se debe usar el método `set`, que recibe el valor como único parámetro:

```
m.set('hola')
```

Para obtener el valor del modelo `m`, se debe usar el método `get`, que no recibe ningún parámetro:

```
s = m.get()
```

En este ejemplo, la variable `s` toma el valor `'hola'`.

Como los modelos creados por `StringVar` almacenan datos de tipo string, hay que tener cuidado de hacer las conversiones apropiadas si se desea usar datos numéricos:

```
a = StringVar()
b = StringVar()
a.set(5)                # es convertido a string
b.set(8)                # es convertido a string
print a.get() + b.get()             # imprime 58
print int(a.get()) + int(b.get())   # imprime 13
```

Usted podría preguntarse cuál es la razón para usar modelos en vez de usar las variables propias de Python, —es decir, las que son creadas mediante asignaciones— para almacenar los datos. Los modelos tienen la ventaja que es posible asociarlos a elementos de la interfaz que responden automáticamente cuando el valor del modelo cambia.

Por ejemplo, podemos asociar una etiqueta a un modelo. La etiqueta siempre mostrará en la interfaz el valor que tiene el modelo, incluso cuando éste cambie. El parámetro `textvariable` asocia el modelo a la etiqueta:

```
x = StringVar()
l = Label(w, textvariable=x)
l.pack()
```

Cada vez que cambie el valor del modelo `x`, el texto de la etiqueta será actualizado inmediatamente.

También podemos asociar un campo de entrada a un modelo. El valor asociado al modelo siempre será el texto que está ingresado en el campo.

Para asociar un modelo a un campo de texto, también se usa el parámetro `textvariable`:

```
x = StringVar()
e = Entry(w, textvariable=x)
e.pack()
```

Cuando se obtenga el valor del modelo mediante la llamada `x.get()`, el valor retornado será lo que el usuario haya ingresado en el campo hasta ese momento.

20.5 Resumen

Para diseñar un programa que tiene una interfaz gráfica, hay tres elementos importantes que hay que tener en consideración.

1. Los elementos que componen la interfaz (la **vista** del programa).
2. Los **modelos** que mantienen el estado de la interfaz en todo momento.
3. Los **controladores** que reaccionan a eventos del usuario.

Los controladores pueden interactuar con los modelos mediante sus métodos `get` y `set`. Los cambios en los modelos pueden verse reflejados en la vista.

Parte II

Ejercicios

Capítulo 21

Expresiones y programas simples

21.1 Evaluación de expresiones

Sin usar el computador, evalúe las siguientes expresiones, y para cada una de ellas indique el resultado y su tipo (si la expresión es válida) o qué error ocurre (si no lo es):

```
>>> 2 + 3       # Respuesta: tipo int, valor 5
>>> 4 / 0       # Respuesta: error de division por cero
>>> 5 + 3 * 2
>>> '5' + '3' * 2
>>> 2 ** 10 == 1000 or 2 ** 7 == 100
>>> int("cuarenta")
>>> 70/16 + 100/24
>>> 200 + 19%
>>> 3 < (1024 % 10) < 6
>>> 'six' + 'eight'
>>> 'six' * 'eight'
>>> float(-int('5') + int('10'))
>>> abs(len('ocho') - len('cinco'))
>>> bool(14) or bool(-20)
>>> float(str(int('5' * 4) / 3)[2])
```

Compruebe sus respuestas en el computador.

21.2 Saludo

Escriba un programa que pida al usuario que escriba su nombre, y lo salude llamándolo por su nombre.

```
Ingrese su nombre: Perico
Hola, Perico
```

21.3 Círculos

Escriba un programa que reciba como entrada el radio de un círculo y entregue como salida su perímetro y su área:

```
Ingrese el radio: 5
Perimetro: 31.4
Area: 78.5
```

21.4 Promedio

Escriba un programa que calcule el promedio de 4 notas ingresadas por el usuario:

```
Primera nota: 55
Segunda nota: 71
Tercera nota: 46
Cuarta nota: 87
El promedio es: 64.75
```

21.5 Conversión de unidades de longitud

Escriba un programa que convierta de centímetros a pulgadas. Una pulgada es igual a 2,54 centímetros.

```
Ingrese longitud: 45
45 cm = 17.7165 in
```

```
Ingrese longitud: 13
13 cm = 5.1181 in
```

21.6 Número invertido

Escriba un programa que pida al usuario un entero de tres dígitos, y entregue el número con los dígitos en orden inverso:

```
Ingrese numero: 345
543
```

```
Ingrese numero: 241
142
```

21.7 Hora futura

Escriba un programa que pregunte al usuario la hora actual t del reloj y un número entero de horas h, que indique qué hora marcará el reloj dentro de h horas:

```
Hora actual: 3
Cantidad de horas: 5
En 5 horas, el reloj marcara las 8
```

```
Hora actual: 11
Cantidad de horas: 43
En 43 horas, el reloj marcara las 6
```

21.8 Parte decimal

Escriba un programa que entregue la parte decimal de un número real ingresado por el usuario.

```
Ingrese un numero: 4.5
0.5
```

```
Ingrese un numero: -1.19
0.19
```

Capítulo 22

Estructuras condicionales

22.1 Determinar par

Escriba un programa que determine si el número entero ingresado por el usuario es par o no.

```
Ingrese un numero: 4
Su numero es par
```

```
Ingrese un numero: 3
Su numero es impar
```

22.2 Años bisiestos

Cuando la Tierra completa una órbita alrededor del Sol, no han transcurrido exactamente 365 rotaciones sobre sí misma, sino un poco más. Más precisamente, la diferencia es de más o menos un cuarto de día.

Para evitar que las estaciones se desfasen con el calendario, el calendario juliano introdujo la regla de introducir un día adicional en los años divisibles por 4 (llamados bisiestos), para tomar en consideración los cuatro cuartos de día acumulados.

Sin embargo, bajo esta regla sigue habiendo un desfase, que es de aproximadamente 3/400 de día.

Para corregir este desfase, en el año 1582 el papa Gregorio XIII introdujo un nuevo calendario, en el que el último año de cada siglo dejaba de ser bisiesto, a no ser que fuera divisible por 400.

Escriba un programa que indique si un año es bisiesto o no, teniendo en cuenta cuál era el calendario vigente en ese año:

```
Ingrese un anno: 1988
1988 es bisiesto
```

```
Ingrese un anno: 2011
2011 no es bisiesto
```

```
Ingrese un anno: 1700
1700 no es bisiesto
```

```
Ingrese un anno: 1500
1500 es bisiesto
```

```
Ingrese un anno: 2400
2400 es bisiesto
```

22.3 Palabra más larga

Escriba un programa que pida al usuario dos palabras, y que indique cuál de ellas es la más larga y por cuántas letras lo es.

```
Palabra 1: edificio
Palabra 2: tren
La palabra edificio tiene 4 letras mas que tren.
```

```
Palabra 1: sol
Palabra 2: paralelepipedo
La palabra paralelepipedo tiene 11 letras mas que sol
```

```
Palabra 1: plancha
Palabra 2: lapices
Las dos palabras tienen el mismo largo
```

22.4 Ordenamiento

Escriba un programa que reciba como entrada dos números, y los muestre ordenados de menor a mayor:

```
Ingrese numero: 51
Ingrese numero: 24
24 51
```

A continuación, escriba otro programa que haga lo mismo con tres números:

```
Ingrese numero: 8
Ingrese numero: 1
Ingrese numero: 4
1 4 8
```

Finalmente, escriba un tercer programa que ordene cuatro números:

```
Ingrese numero: 7
Ingrese numero: 0
Ingrese numero: 6
Ingrese numero: 1
0 1 6 7
```

Recuerde que su programa debe entregar la solución correcta para cualquier combinación de números, no sólo para los ejemplos mostrados aquí.

Hay más de una manera de resolver cada ejercicio.

22.5 Edad

Escriba un programa que entregue la edad del usuario a partir de su fecha de nacimiento:

```
Ingrese su fecha de nacimiento.
Dia: 14
Mes: 6
Anno: 1948
Usted tiene 62 annos
```

Por supuesto, el resultado entregado depende del día en que su programa será ejecutado.

Para obtener la fecha actual, puede hacerlo usando la función `localtime` que viene en el módulo time. Los valores se obtienen de la siguiente manera (suponiendo que hoy es 11 de marzo de 2011):

```
>>> from time import localtime
>>> t = localtime()
>>> t.tm_mday
11
>>> t.tm_mon
3
>>> t.tm_year
2011
```

El programa debe tener en cuenta si el cumpleaños ingresado ya pasó durante este año, o si todavía no ocurre.

22.6 Set de tenis

El joven periodista Solarrabietas debe relatar un partido de tenis, pero no conoce las reglas del deporte. En particular, no ha logrado aprender cómo saber si un set ya terminó, y quién lo ganó.

Un partido de tenis se divide en sets. Para ganar un set, un jugador debe ganar 6 juegos, pero además debe haber ganado por lo menos dos juegos más que su rival. Si el set está empatado a 5 juegos, el ganador es el primero que

llegue a 7. Si el set está empatado a 6 juegos, el set se define en un último juego, en cuyo caso el resultado final es 7-6.

Sabiendo que el jugador A ha ganado *m* juegos, y el jugador B, *n* juegos, al periodista le gustaría saber:

- si A ganó el set, o
- si B ganó el set, o
- si el set todavía no termina, o
- si el resultado es inválido (por ejemplo, 8-6 o 7-3).

Desarrolle un programa que solucione el problema de Solarrabietas:

```
Juegos ganados por A: 4
Juegos ganados por B: 5
Aun no termina
```

```
Juegos ganados por A: 5
Juegos ganados por B: 7
Gano B
```

```
Juegos ganados por A: 5
Juegos ganados por B: 6
Aun no termina
```

```
Juegos ganados por A: 3
Juegos ganados por B: 7
Invalido
```

```
Juegos ganados por A: 6
Juegos ganados por B: 4
Gano A
```

22.7 Triángulos

Los tres lados *a*, *b* y *c* de un triángulo deben satisfacer la desigualdad triangular: cada uno de los lados no puede ser más largo que la suma de los otros dos.

Escriba un programa que reciba como entrada los tres lados de un triángulo, e indique:

- si acaso el triángulo es inválido; y
- si no lo es, qué tipo de triángulo es.

```
Ingrese a: 3.9
Ingrese b: 6.0
Ingrese c: 1.2
No es un triangulo valido.
```

```
Ingrese a: 1.9
Ingrese b: 2
Ingrese c: 2
El triangulo es isoceles.
```

```
Ingrese a: 3.0
Ingrese b: 5.0
Ingrese c: 4.0
El triangulo es escaleno.
```

Capítulo 23

Ciclos

23.1 Múltiplos

Escriba un programa que muestre la tabla de multiplicar del 1 al 10 del número ingresado por el usuario:

```
Ingrese un numero: 9
9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
9 x 10 = 90
```

23.2 Potencias de dos

Escriba un programa que genere todas las potencias de 2, desde la 0-ésima hasta la ingresada por el usuario:

```
Ingrese num: 10
1 2 4 8 16 32 64 128 256 512 1024
```

23.3 Divisores

Escriba un programa que entregue todos los divisores del número entero ingresado:

```
Ingrese numero: 200
1 2 4 5 8 10 20 25 40 50 100 200
```

23.4 Tabla de multiplicar

Escriba un programa que muestre una tabla de multiplicar como la siguiente. Los números deben estar alineados a la derecha.

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

23.5 Tiempo de viaje

Un viajero desea saber cuánto tiempo tomó un viaje que realizó. Él tiene la duración en minutos de cada uno de los tramos del viaje.

Desarrolle un programa que permita ingresar los tiempos de viaje de los tramos y entregue como resultado el tiempo total de viaje en formato `horas:minutos`. El programa debe dejar de pedir tiempos de viaje cuando se ingresa un 0.

```
Duracion tramo: 15
Duracion tramo: 30
Duracion tramo: 87
Duracion tramo: 0
Tiempo total de viaje: 2:12 horas
```

```
Duracion tramo: 51
Duracion tramo: 17
Duracion tramo: 0
Tiempo total de viaje: 1:08 horas
```

23.6 Dibujos de asteriscos

Escriba un programa que pida al usuario ingresar la altura y el ancho de un rectángulo y lo dibuje utilizando asteriscos:

```
Altura: 3
Ancho: 5

*****
*****
*****
```

Escriba un programa que dibuje el triángulo del tamaño indicado por el usuario de acuerdo al ejemplo:

```
Altura: 5

*
**
***
****
*****
```

Escriba un programa que dibuje el hexágono del tamaño indicado por el usuario de acuerdo al ejemplo:

```
Lado: 4

   ****
  ******
 ********
**********
 ********
  ******
   ****
```

23.7 π

Desarolle un programa para estimar el valor de π usando la siguiente suma infinita:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right)$$

La entrada del programa debe ser un número entero n que indique cuántos términos de la suma se utilizará.

```
n: 3
3.466666666666667
```

```
n: 1000
3.140592653839794
```

23.8 e

El número de Euler, $e \approx 2,71828$, puede ser representado como la siguiente suma infinita:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$$

Desarrolle un programa que entregue un valor aproximado de e, calculando esta suma hasta que la diferencia entre dos sumandos consecutivos sea menor que 0,0001.

Recuerde que el factorial $n!$ es el producto de los números de 1 a n.

23.9 Secuencia de Collatz

La secuencia de Collatz de un número entero se construye de la siguiente forma:

- si el número es par, se lo divide por dos;
- si es impar, se le multiplica tres y se le suma uno;
- la sucesión termina al llegar a uno.

La conjetura de Collatz afirma que, al partir desde cualquier número, la secuencia siempre llegará a 1. A pesar de ser una afirmación a simple vista muy simple, no se ha podido demostrar si es cierta o no.

Usando computadores, se ha verificado que la sucesión efectivamente llega a 1 partiendo desde cualquier número natural menor que 2^{58}.

Desarrolle un programa que entregue la secuencia de Collatz de un número entero:

```
n: 18
18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

```
n: 19
19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

```
n: 20
20 10 5 16 8 4 2 1
```

A continuación, desarrolle un programa que grafique los largos de las secuencias de Collatz de los números enteros positivos menores que el ingresado por el usuario:

```
n: 10
1 *
2 **
3 ********
4 ***
5 ******
6 *********
7 ****************
8 ****
9 ********************
10 *******
```

Capítulo 24

Patrones comunes

24.1 Número mayor

Escriba un programa que permita determinar el número mayor perteneciente a un conjunto de n números, donde tanto el valor de n como el de los números deben ser ingresados por el usuario.

```
Ingrese n: 4
Ingrese numero: 23
Ingrese numero: -34
Ingrese numero: 0
Ingrese numero: 1
El mayor es 23
```

24.2 Productos especiales

Escriba sendos programas que pidan al usuario la entrada correspondiente y calculen las siguientes operaciones:

1. El factorial $n!$ de un número entero $n \geq 0$, definido como:

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n.$$

   Además, se define $0! = 1$.

2. La potencia factorial creciente $n^{\overline{m}}$, definida como:

$$n^{\overline{m}} = n(n+1)(n+2)\cdots(n+m-1).$$

3. El coeficiente binomial $\binom{n}{k}$, definido como:

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-k+1)}{1 \cdot 2 \cdot 3 \cdot \cdots \cdot k} = \frac{n!}{(n-k)!\,k!}.$$

4. El número de Stirling del segundo tipo $S(n,k)$, que se puede calcular como:

$$S(n,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^j\binom{k}{i}(k-i)^n.$$

24.3 Contar combinaciones de dados

Un jugador debe lanzar dos dados numerados de 1 a 6, y su puntaje es la suma de los valores obtenidos.

Un puntaje dado puede ser obtenido con varias combinaciones posibles. Por ejemplo, el puntaje 4 se logra con las siguientes tres combinaciones: $1+3$, $2+2$ y $3+1$.

Escriba un programa que pregunte al usuario un puntaje, y muestre como resultado la cantidad de combinaciones de dados con las que se puede obtener ese puntaje.

```
Ingrese el puntaje: 4
Hay 3 combinaciones para obtener 4
```

```
Ingrese el puntaje: 11
Hay 2 combinaciones para obtener 11
```

```
Ingrese el puntaje: 17
Hay 0 combinaciones para obtener 17
```

24.4 Histograma

Escriba un programa que pida al usuario que ingrese varios valores enteros, que pueden ser positivos o negativos. Cuando se ingrese un cero, el programa debe terminar y mostrar un gráfico de cuántos valores positivos y negativos fueron ingresados:

```
Ingrese varios valores, termine con cero:
-17
-12
14
-5
-8128
3
-2
-9
1500
-43
0
Positivos: ***
Negativos: *******
```

24.5 Más corta y más larga

Desarrolle un programa que pregunte al usuario un entero *n* y a continuación le pida ingresar *n* palabras. La salida del programa debe mostrar cuáles fueron la palabra más larga y la más corta. Recuerde que la función `len` entrega el largo de un string:

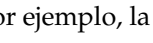
```
>>> len('amarillo')
8
```

La ejecución del programa debe verse así:

```
Cantidad de palabras: 5
Palabra 1: negro
Palabra 2: amarillo
Palabra 3: naranjo
Palabra 4: azul
Palabra 5: blanco
La palabra mas larga es amarillo
La palabra mas corta es azul
```

24.6 Piezas de dominó

Desarrolle un programa que permita determinar la cantidad total de puntos que contiene un juego de dominó de 28 piezas.

Las piezas de dominó tienen dos lados, cada uno de los cuales tiene entre cero y seis puntos. Por ejemplo, la pieza  tiene cinco puntos en total.

Capítulo 25

Ruteos

25.1 Ojo con la indentación

Sin usar el computador, rutee los siguientes tres programas e indique cuál es la salida de cada uno de ellos. Compruebe que sus respuestas son correctas ejecutando los programas en el computador.

1. ```
   s = 0
   t = 0
   for i in range(3):
       for j in range(3):
           s = s + 1
           if i < j:
               t = t + 1
       print t
       print s
   ```

2. ```
   s = 0
   t = 0
   for i in range(3):
       for j in range(3):
           s = s + 1
       if i < j:
           t = t + 1
           print t
   print s
   ```

3. ```
   s = 0
   t = 0
   for i in range(3):
       for j in range(3):
           s = s + 1
   ```
   ```

```
if i < j:
    t = t + 1
    print t
print s
```

25.2 Ruteos varios

Solácese ruteando los siguientes programas.

1. ```
   j = 2
   c = 1
   p = True
   while j > 0:
       j = j - c
       if p:
           c = c + 1
       p = not p
   print j < 0 and p
   ```

2. ```
   a = 10
   c = 1
   x = 0
   y = x + 1
   z = y
   while z <= y:
       z = z + c
       y = 2 * x + z
       if y < 4:
           y = y + 1
       x = x - 1
   print x, y, z
   ```

3. ```
   a = 11
   b = a / 3
   c = a / 2
   n = 0
   while a == b + c:
       n += 1
       b += c
       c = b - c
       if n % 2 == 0:
           a = 2 * a - 3
       print 100 * b + c
   ```
   ```

Capítulo 26

Diseño de algoritmos

26.1 Dígitos

Escriba un programa que determine la cantidad de dígitos en un número entero ingresado por el usuario:

```
Ingrese numero: 2048
2048 tiene 4 digitos
```

```
Ingrese numero: 12
12 tiene 2 digitos
```

```
Ingrese numero: 0
0 tiene 1 digito
```

26.2 Ecuación primer grado

Escriba un programa que pida los coeficientes de una ecuación de primer grado $ax + b = 0$. y que entregue la solución.

Una ecuación de primer grado puede tener solución única, tener infinitas soluciones o no tener soluciones.

```
Ingrese a: 0
Ingrese b: 3

Sin solucion
```

```
Ingrese a: 4
Ingrese b: 2

Solucion unica: -0.5
```

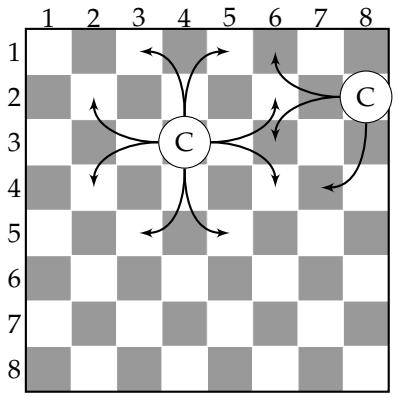


Figura 26.1: Ejemplos de los movimientos posibles del caballo de ajedrez.

```
Ingrese a: 0
Ingrese b: 0

No hay solucion unica.
```

26.3 Caballo de ajedrez

Un tablero de ajedrez es una grilla de 8×8 casillas. Cada celda puede ser representada mediante las coordenadas de su fila y su columna, numeradas desde 1 hasta 8.

El caballo es una pieza que se desplaza en forma de L: su movimiento consiste en avanzar dos casillas en una dirección y luego una casilla en una dirección perpendicular a la primera, como se muestra en la figura 26.1.

Escriba un programa que reciba como entrada las coordenadas en que se encuentra un caballo, y entregue como salida todas las casillas hacia las cuales el caballo puede desplazarse.

Todas las coordenadas mostradas deben estar dentro del tablero.

Si la coordenada ingresada por el usuario es inválida, el programa debe indicarlo.

```
Ingrese coordenadas del caballo.
Fila: 2
Columna: 8

El caballo puede saltar de 2 8 a:
1 6
3 6
4 7
```

```
Ingrese coordenadas del caballo.
Fila: 3
Columna: 4

El caballo puede saltar de 3 4 a:
1 3
1 5
2 2
2 6
4 2
4 6
5 3
5 5
```

```
Ingrese coordenadas del caballo.
Fila: 1
Columna: 9

Posicion invalida.
```

26.4 Media armónica

La media armónica de una secuencia de n números reales $x_1, x_2, \ldots, x_n$ se define como:

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \frac{1}{x_3} + \cdots + \frac{1}{x_n}}.$$

Desarrolle un programa que calcule la media armónica de una secuencia de números.

El programa primero debe preguntar al usuario cuántos números ingresará. A continuación, pedirá al usuario que ingrese cada uno de los n números reales. Finalmente, el programa mostrará el resultado.

```
Cuantos numeros: 3
n1 = 1
n2 = 3
n3 = 2
H = 1.63636363636363636363
```

26.5 Números palíndromos

Un número natural es un palíndromo si se lee igual de izquierda a derecha y de derecha a izquierda. Por ejemplo, 14941 es un palíndromo, mientras que 81924 no lo es. Escriba un programa que indique si el número ingresado es o no palíndromo:

```
Ingrese un numero: 14941
14941 es palindromo
```

```
Ingrese un numero: 81924
81924 no es palindromo
```

26.6 Palabras palíndromas

Así como hay números palíndromos, también hay palabras palíndromas, que son las que no cambian al invertir el orden de sus letras. Por ejemplo, «reconocer», «Neuquén» y «acurruca» son palíndromos.

1. Escriba un programa que reciba como entrada una palabra e indique si es palíndromo o no. Para simplificar, suponga que la palabra no tiene acentos y todas sus letras son minúsculas:

```
Ingrese palabra: sometemos
Es palindromo
```

```
Ingrese palabra: rascar
No es palindromo
```

2. Modifique su programa para que reconozca oraciones palíndromas. La dificultad radica en que hay que ignorar los espacios:

```
Ingrese oracion: dabale arroz a la zorra el abad
Es palindromo
```

```
Ingrese oracion: eva usaba rimel y le miraba suave
Es palindromo
```

```
Ingrese oracion: puro chile es tu cielo azulado
No es palindromo
```

26.7 Cachipún

En cada ronda del juego del cachipún, los dos competidores deben elegir entre jugar tijera, papel o piedra.

Las reglas para decidir quién gana la ronda son: tijera le gana a papel, papel le gana a piedra, piedra le gana a tijera, y todas las demás combinaciones son empates.

El ganador del juego es el primero que gane tres rondas.

Escriba un programa que pregunte a cada jugador cuál es su jugada, muestre cuál es el marcador después de cada ronda, y termine cuando uno de ellos haya ganado tres rondas. Los jugadores deben indicar su jugada escribiendo `tijera`, `papel` o `piedra`.

```
A: tijera
B: papel
1 - 0

A: tijera
B: tijera
1 - 0

A: piedra
B: papel
1 - 1

A: piedra
B: tijera
2 - 1

A: papel
B: papel
2 - 1

A: papel
B: piedra
3 - 1

A es el ganador
```

26.8 Números primos

Un número primo es un número natural que sólo es divisible por 1 y por sí mismo. Los números que tienen más de un divisor se llaman números compuestos. El número 1 no es ni primo ni compuesto.

1. Escriba un programa que reciba como entrada un número natural, e indique si es primo o compuesto.

2. Escriba un programa que muestre los n primeros números primos, donde n es ingresado por el usuario.

3. Escriba un programa que muestre los números primos menores que m, donde m es ingresado por el usuario.

4. Escriba un programa que cuente cuántos son los números primos menores que m, donde m es ingresado por el usuario.

5. Todos los números naturales mayores que 1 pueden ser factorizados de manera única como un producto de divisores primos.

Escriba un programa que muestre los factores primos de un número entero ingresado por el usuario:

```
Ingrese numero: 204
2
2
3
17
```

6. La conjetura de Goldbach sugiere que todo número par mayor que dos puede ser escrito como la suma de dos números primos. Hasta ahora no se conoce ningún número para el que esto no se cumpla.

   Escriba un programa que reciba un número par como entrada y muestre todas las maneras en que puede ser escrito como una suma de dos primos:

```
Ingrese numero par: 338
7 + 331
31 + 307
61 + 277
67 + 271
97 + 241
109 + 229
127 + 211
139 + 199
157 + 181
```

   Muestre sólo una de las maneras de escribir cada suma (por ejemplo, si muestra $61 + 271$, no muestre $271 + 61$).

7. Escriba programas que respondan las siguientes preguntas:

- ¿Cuántos primos menores que diez mil terminan en 7?
- ¿Cuál es la suma de los cuadrados de los números primos entre 1 y 1000? (Respuesta: 49345379).
- ¿Cuál es el producto de todos los números primos menores que 100 que tienen algún dígito 7? (Respuesta: $7 \times 17 \times 37 \times 47 \times 67 \times 71 \times 73 \times 79 \times 97 = 550682633299463$).

26.9 El mejor número

Según Sheldon[1], el mejor número es el 73.

73 es el 21er número primo. Su espejo, 37, es el 12mo número primo. 21 es el producto de multiplicar 7 por 3. En binario, 73 es un palíndromo: 1001001.

Escriba programas que le permitan responder las siguientes preguntas:

[1] http://www.youtube.com/watch?v=Gg9kSn3NRVk

1. ¿Existen otros valores p que sean el n-ésimo primo, tales que espejo(p) es el espejo(n)-ésimo primo?

2. ¿Existen otros valores p que sean el n-ésimo primo, tales que n es el producto de los dígitos de p?

3. ¿Cuáles son los primeros diez números primos cuya representación binaria es un palíndromo?

26.10 Adivinar el número

Escriba un programa que «piense» un número entre 0 y 100, y entregue pistas al usuario para que lo adivine.

El programa puede obtener un número al azar entre 0 y 100 de la siguiente manera (¡haga la prueba!):

```
>>> from random import randrange
>>> n = randrange(101)
>>> print n
72
```

El usuario debe ingresar su intento, y el programa debe decir si el número pensado es mayor, menor, o el correcto:

```
Adivine el numero.
Intento 1: 32
Es mayor que 32
Intento 2: 80
Es menor que 80
Intento 3: 70
Es mayor que 70
Intento 4: 72
Correcto. Adivinaste en 4 intentos.
```

Una vez que complete ese ejercicio, es hora de invertir los roles: ahora usted pensará un número y el computador lo adivinará.

Escriba un programa que intente adivinar el número pensado por el usuario. Cada vez que el computador haga un intento, el usuario debe ingresar <, > o =, dependiendo si el intento es menor, mayor o correcto.

La estrategia que debe seguir el programa es recordar siempre cuáles son el menor y el mayor valor posibles, y siempre probar con el valor que está en la mitad. Por ejemplo, si usted piensa el número 82, y no hace trampa al jugar, la ejecución del programa se verá así:

```
Intento 1: 50
>
Intento 2: 75
>
Intento 3: 88
```

n	0	1	2	3	4	5	6	7	8	9	10	11	12	...
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144	...

Tabla 26.1: Los primeros términos F_n de la sucesión de Fibonacci.

```
<
Intento 4: 81
>
Intento 5: 84
<
Intento 6: 82
=
Adivine en 6 intentos B-)
```

26.11 Números de Fibonacci

Los números de Fibonacci F_k son una sucesión de números naturales definidos de la siguiente manera:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_k &= F_{k-1} + F_{k-2} \qquad \text{cuando } k \geq 2. \end{aligned}$$

En palabras simples, la sucesión de Fibonacci comienza con 0 y 1, y los siguientes términos siempre son la suma de los dos anteriores.

En la tabla 26.1 podemos ver los números de Fibonacci desde el 0-ésimo hasta el duodécimo.

1. Escriba un programa que reciba como entrada un número entero n, y entregue como salida el n-ésimo número de Fibonacci:

```
Ingrese n: 11
F11 = 89
```

2. Escriba un programa que reciba como entrada un número entero e indique si es o no un número de Fibonacci:

```
Ingrese un numero: 34
34 es numero de Fibonacci
```

```
Ingrese un numero: 78
78 no es numero de Fibonacci
```

3. Escriba un programa que muestre los m primeros números de Fibonacci, donde m es un número ingresado por el usuario:

Estimación y	Cuociente x/y	Promedio
1	$2/1 = 2$	$(2+1)/2 = 1{,}5$
1,5	$2/1{,}5 = 1{,}3333$	$(1{,}3333+1{,}5)/2 = 1{,}4167$
1,4167	$2/1{,}4167 = 1{,}4118$	$(1{,}4118+1{,}4167)/2 = 1{,}4142$
1,4142	...	...

Tabla 26.2: Primeras iteraciones del método de Newton para aproximar la raíz cuadrada de 2 usando la aproximación inicial $y = 1$.

```
Ingrese m: 7
Los 7 primeros numeros de Fibonacci son:
0 1 1 2 3 5 8
```

26.12 Espiral

Ejercicio sacado de Project Euler[2].

La siguiente espiral de 5×5 se forma comenzando por el número 1, y moviéndose a la derecha en el sentido de las agujas del reloj:

21	22	23	24	**25**
20	**7**	8	**9**	10
19	6	**1**	2	11
18	**5**	4	**3**	12
17	16	15	14	**13**

La suma de las diagonales de esta espiral es 101.

Escriba un programa que entregue la suma de las diagonales de una espiral de 1001×1001 creada de la misma manera.

26.13 Método de Newton

Ejercicio sacado de SICP[3].

El método computacional más común para calcular raíces cuadradas (y otras funciones también) es el método de Newton de aproximaciones sucesivas. Cada vez que tenemos una estimación y del valor de la raíz cuadrada de un número x, podemos hacer una pequeña manipulación para obtener una mejor aproximación (una más cercana a la verdadera raíz cuadrada) promediando y con x/y.

La tabla 26.2 muestra cómo refinar paso a paso la raíz cuadrada de dos a partir de la aproximación inicial $\sqrt{2} \approx 1$.

[2] Publicado en `http://projecteuler.net/problem=28` bajo licencia Creative Commons BY-NC-SA 2.0.

[3] Harold Abelson, Gerald Jay Sussman y Julie Sussman. *Structure and Interpretation of Computer Programs*, 2da edición. Publicado en `http://mitpress.mit.edu/sicp/` bajo licencia Creative Commons BY-SA 3.0. ¡Este libro es un clásico!

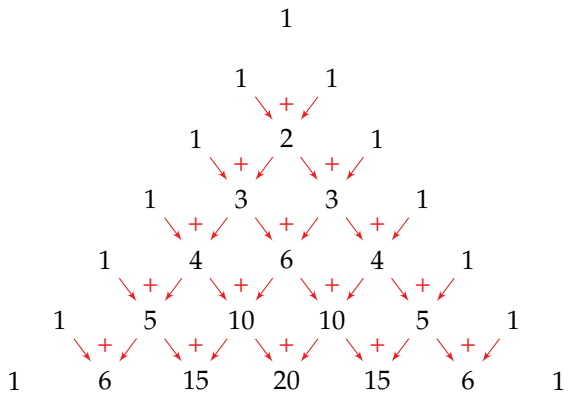


Figura 26.2: Triángulo de Pascal

Al continuar este proceso, obtenemos cada vez mejores estimaciones de la raíz cuadrada.

El algoritmo debe detenerse cuando la estimación es «suficientemente buena», que puede ser cualquier criterio bien definido.

1. Escriba un programa que reciba como entrada un número real x y calcule su raíz cuadrada usando el método de Newton. El algoritmo debe detenerse cuando el cuadrado de la raíz cuadrada estimada difiera de x en menos de 0,0001.

   (Este criterio de detención no es muy bueno).

2. Escriba un programa que reciba como entrada el número real x y un número entero indicando con cuántas cifras decimales de precisión se desea obtener su raíz cuadrada.

   El método de Newton debe detenerse cuando las cifras de precisión deseadas no cambien de una iteración a la siguiente.

   Por ejemplo, para calcular $\sqrt{2}$ con dos cifras de precisión, las estimaciones sucesivas son 1; 1,5; 1,416667 y 1,414216. El algoritmo debe detenerse en la cuarta iteración, pues en ella las dos primeras cifras decimales no cambiaron con respecto a la iteración anterior:

   ```
   Ingrese x: 2
   Cifras decimales: 2
   La raiz es 1.4142156862745097
   ```

   (La cuarta aproximación es bastante cercana a la verdadera raíz 1,4142135623730951).

26.14 Triángulo de Pascal

Desarrolle un programa que dibuje un triángulo de Pascal, o sea, una disposición de números enteros tales que cada uno sea la suma de los dos que están por encima de él, tal como se ve en la figura 26.2.

Un programa que genere las primeras cinco líneas podría verse así:

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5 10 10  5  1
```

o así:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
```

Usted, empero, debe generar las primeras veinte líneas. Considere que en la línea 20 aparecen números de cinco dígitos.

Capítulo 27

Funciones y módulos

27.1 Número par

Escriba la función `par(x)` que retorne **`True`** si `x` es par, y **`False`** si es impar:

```
>>> par(16)
True
>>> par(29)
False
>>> par('hola')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 2, in par
TypeError: not all arguments converted during string formatting
```

27.2 Números palíndromos

Escriba la función `invertir_digitos(n)` que reciba un número entero `n` y entregue como resultado el número `n` con los dígitos en el orden inverso:

```
>>> invertir_digitos(142)
241
```

A continuación, escriba un programa que indique si el número ingresado es palíndromo o no, usando la función `invertir_digitos`:

```
Ingrese n: 81418
Es palindromo
```

27.3 Funciones de números primos

Para el ejercicio 26.8 usted debió desarrollar programas sobre números primos. Muchos de estos programas sólo tenían pequeñas diferencias entre ellos,

por lo que había que repetir mucho código al escribirlos. En este ejercicio, usted deberá implementar algunos de esos programas como funciones, reusando componentes para evitar escribir código repetido.

1. Escriba la función `es_divisible(n, d)` que indique si `n` es divisible por `d`:

```
>>> es_divisible(15, 5)
True
>>> es_divisible(15, 6)
False
```

2. Usando la función `es_divisible`, escriba una función `es_primo(n)` que determine si el número `n` es primo o no:

```
>>> es_primo(17)
True
>>> es_primo(221)
False
```

3. Usando la función `es_primo`, escriba la función `i_esimo_primo(i)` que entregue el `i`-ésimo número primo:

```
>>> i_esimo_primo(1)
2
>>> i_esimo_primo(20)
71
```

4. Usando las funciones anteriores, escriba la función `primeros_primos(m)` que entregue una lista de los primeros `m` números primos:

```
>>> primeros_primos(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

5. Usando las funciones anteriores, escriba la función `primos_hasta(m)` que entregue una lista de los primos menores o iguales que `m`:

```
>>> primos_hasta(19)
[2, 3, 5, 7, 11, 13, 17, 19]
```

6. Cree un módulo llamado `primos.py` que contenga todas las funciones anteriores. Al ejecutar `primos.py` como un programa por sí solo, debe mostrar, a modo de prueba, los veinte primeros números primos. Al importarlo como un módulo, esto no debe ocurrir.

7. Un *primo de Mersenne* es un número primo de la forma $2^p - 1$. Una propiedad conocida de los primos de Mersenne es que p debe ser también un número primo.

   Escriba un programa llamado `mersenne.py` que pregunte al usuario un entero n, y muestre como salida los primeros n primos de Mersenne:

```
Cuantos primos de Mersenne: 5
3
7
31
127
8191
```

Su programa debe importar el módulo `primos` y usar las funciones que éste contiene.

27.4 Aproximación de seno y coseno

La funciones seno y coseno puede ser representadas mediante sumas infinitas:

$$\text{sen}(x) = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$\cos(x) = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

(Éstas son las series de Taylor en torno a $x = 0$ de las funciones seno y coseno, que usted estudiará en Matemáticas 2).

Los términos de ambas sumas son cada vez más pequeños, por lo que tomando algunos de los primeros términos es posible obtener una buena aproximación.

1. Escriba la función `factorial_reciproco(n)`, que retorne el valor $1/n!$

2. Escriba la función `signo(n)` que retorne 1 cuando `n` es par y -1 cuando `n` es impar.

3. Escriba las funciones `seno_aprox(x, m)` y `coseno_aprox(x, m)` que aproximen respectivamente el seno y el coseno usando los `m` primeros términos de las sumas correspondientes. Las funciones deben llamar a las funciones `factorial_reciproco` y `signo`.

4. Escriba la función `error(f_exacta, f_aprox, m, x)` que entregue cuál es la diferencia entre el valor exacto de la función `f_exacta` y su aproximación con `m` términos usando la función `f_aprox` en $x =$ `x`.

   Por ejemplo, el error del seno en $x = 2$ al usar 20 términos se obtendría así:

   ```
   >>> from math import sin
   >>> error(sin, seno_aprox, 20, 2)
   ```

27.5 Tabla de verdad

Un *predicado lógico* es una función cuyos parámetros son booleanos y su resultado también es booleano.

Escriba la función `tabla_de_verdad(predicado)` que reciba como parámetro un predicado lógico de tres parámetros e imprima la tabla de verdad del predicado.:

```
>>> def predicado(p, q, r):
...     return (not p) and (q or r)
...
>>> tabla_verdad(predicado)
p     q     r     predicado
===== ===== ===== =========
True  True  True  False
True  True  False False
True  False True  False
True  False False False
False True  True  True
False True  False True
False False True  True
False False False False
```

Note que la función `tabla_verdad` no retorna nada, sólo imprime la tabla.

27.6 Máximo común divisor

Escriba la función `mcd(a, b)` que entrege el máximo común divisor de los enteros `a` y `b`:

La manera obvia de implementar este programa es literalmente buscando el mayor de los divisores comunes. Existe una técnica más eficiente, que es conocida como *algoritmo de Euclides*[1]. Este método tiene importancia histórica, ya que es uno de los algoritmos más antiguos que aún sigue siendo utilizado.

Resuelva este problema de las dos maneras.

27.7 Módulo de listas

Desarrolle un módulo llamado `listas.py` que contenga las siguientes funciones.

- Una función `promedio(l)`, cuyo parámetro `l` sea una lista de números reales, y que entregue el promedio de los números:

[1] `http://es.wikipedia.org/wiki/Algoritmo_de_Euclides`

```
>>> promedio([7.0, 3.1, 1.7])
3.933333333333333
>>> promedio([1, 4, 9, 16])
7.5
```

- Una función `cuadrados(l)`, que entregue una lista con los cuadrados de los valores de `l`:

```
>>> cuadrados([1, 2, 3, 4, 5])
[1, 4, 9, 16, 25]
>>> cuadrados([3.4, 1.2])
[11.559999999999999, 1.44]
```

- Una función `mas_largo(palabras)`, cuyo parámetro `palabras` es una lista de strings, que entregue cuál es el string más largo:

```
>>> mas_largo(['raton', 'hipopotamo', 'buey', 'jirafa'])
'hipopotamo'
>>> mas_largo(['****', '**', '********', '**'])
'********'
```

Si las palabras más largas son varias, basta que entregue una de ellas.

27.8 Ruteo de funciones

Rutee los siguientes programas, e indique qué es lo que escriben por pantalla.

1.
```
def f(a, b):
    c = a + 2 * b
    d = b ** 2
    return c + d

a = 3
b = 2
c = f(b, a)
d = f(a, b)
print c, d
```

2.
```
def f(x):
    a = x ** 2
    b = a + g(a)
    return a * b

def g(x):
    a = x * 3
```

```
    return a ** 2

m = f(1)
n = g(1)
print m, n
```

3.
```
def f(n):
    if n == 0 or n == 1:
        return 1
    a = f(n - 2)
    b = f(n - 1)
    s = a + b
    return s

print f(5)
```

Capítulo 28

Tuplas

28.1 Expresiones con tuplas

Considere las siguientes asignaciones:

```
>>> a = (2, 10, 1991)
>>> b = (25, 12, 1990)
>>> c = ('Donald', True, b)
>>> x, y = ((27, 3), 9)
>>> z, w = x
>>> v = (x, a)
```

Sin usar el computador, indique cuál es el resultado y el tipo de las siguientes expresiones. A continuación, verifique sus respuestas en el computador.

- `a < b`
- `y + w`
- `x + a`
- `len(v)`
- `v[1][1]`
- `c[0][0]`
- `z, y`
- `a + b[1:5]`
- `(a + b)[1:5]`
- `str(a[2]) + str(b[2])`
- `str(a[2] + b[2])`
- `str((a + b)[2])`
- `str(a + b)[2]`

28.2 Rectas

Una recta en el plano está descrita por la ecuación $y = mx + b$, donde m es la *pendiente* y b es el *intercepto*. Todos los puntos de la recta satisfacen esta ecuación.

En un programa, una recta puede ser representada como una tupla `(m, b)`.

Los algoritmos para resolver los siguientes ejercicios seguramente usted los aprendió en el colegio. Si no los recuerda, puede buscarlos en su libro de matemáticas favorito o en internet.

1. Escriba la función `punto_en_recta(p, r)` que indique si el punto `p` está en la recta `r`:

```
>>> recta = (2, -1)      # esta es la recta y = 2x - 1
>>> punto_en_recta((2, 3), recta)
True
>>> punto_en_recta((0, -1), recta)
True
>>> punto_en_recta((1, 2), recta)
False
```

2. Escriba la función `son_paralelas(r1, r2)` que indique si las rectas `r1` y `r2` son paralelas, es decir, no se intersectan en ningún punto.

3. Escriba la función `recta_que_pasa_por(p1, p2)` que entregue la recta que pasa por los puntos `p1` y `p2`:

```
>>> recta_que_pasa_por((-2, 4), (4, 1))
(-0.5, 3.0)
```

   Puede comprobar que la función está correcta verificando que ambos puntos están en la recta obtenida:

```
>>> p1 = (-2, 4)
>>> p2 = (4, 1)
>>> r = recta_que_pasa_por(p1, p2)
>>> punto_en_recta(p1, r) and punto_en_recta(p2, r)
True
```

4. Escriba la función `punto_de_interseccion(r1, r2)` que entregue el punto donde las dos rectas se intersectan:

```
>>> r1 = (2, 1)
>>> r2 = (-1, 4)
>>> punto_de_interseccion(r1, r2)
(1.0, 3.0)
```

   Si las rectas son paralelas, la función debe retornar **`None`**.

28.3 Fechas

Una fecha puede ser representada como una tupla `(anno, mes, dia)`.

1. Escriba la función `dia_siguiente(f)` que reciba como parámetro una fecha `f` y entegue cuál es la fecha siguiente:

```
>>> dia_siguiente((2011, 4, 11))
(2011, 4, 12)
>>> dia_siguiente((2011, 4, 30))
(2011, 5, 1)
>>> dia_siguiente((2011, 12, 31))
(2012, 1, 1)
```

Como recomendación, dentro de su función cree una lista con la cantidad de días que tiene cada mes:

```
dias_mes = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

2. Escriba la función `dias_entre(f1, f2)` que retorne la cantidad de días que han transcurrido entre las fechas `f1` y `f2`:

```
>>> hoy = (2011, 4, 11)
>>> navidad = (2011, 12, 25)
>>> dias_entre(hoy, navidad)
258
>>> dias_entre(hoy, hoy)
0
```

3. Escriba un programa que le diga al usuario cuántos días de edad tiene:

```
Ingrese su fecha de nacimiento.
Dia: 14
Mes: 5
Anno: 1990
Ingrese la fecha de hoy.
Dia: 20
Mes: 4
Anno: 2011
Usted tiene 7646 dias de edad
```

28.4 Supermercado

Un supermercado utiliza tablas de datos para llevar la información de su inventario. En un programa, cada tabla de datos es una lista de tuplas.

La lista `productos` tiene el código, el nombre, el precio y la cantidad de unidades del producto en bodega:

```
productos = [
    (41419, 'Fideos',        450, 210),
    (70717, 'Cuaderno',      900, 119),
    (78714, 'Jabon',         730, 708),
    (30877, 'Desodorante',  2190,  79),
    (47470, 'Yogur',          99, 832),
    (50809, 'Palta',         500,  55),
    (75466, 'Galletas',      235,   0),
    (33692, 'Bebida',        700,  20),
    (89148, 'Arroz',         900, 121),
    (66194, 'Lapiz',         120, 900),
    (15982, 'Vuvuzela',    12990,  40),
    (41235, 'Chocolate',    3099,  48),
]
```

La lista `clientes` tiene el rut y el nombre de los clientes del supermercado:

```
clientes = [
    ('11652624-7', 'Perico Los Palotes'),
    ( '8830268-0', 'Leonardo Farkas'),
    ( '7547896-8', 'Fulanita de Tal'),
]
```

La lista `ventas` contiene las ventas realizadas, representadas por el número de boleta, la fecha de la venta y el rut del cliente:

```
ventas = [
    (1, (2010,  9, 12),  '8830268-0'),
    (2, (2010,  9, 19), '11652624-7'),
    (3, (2010,  9, 30),  '7547896-8'),
    (4, (2010, 10,  1),  '8830268-0'),
    (5, (2010, 10, 13),  '7547896-8'),
    (6, (2010, 11, 11), '11652624-7'),
]
```

El detalle de cada venta se encuentra en la lista `itemes`. Cada ítem tiene asociado un número de boleta, un código de producto y una cantidad:

```
itemes = [
    (1, 89148,  3), (2, 50809,  4), (2, 33692,  2),
    (2, 47470,  6), (3, 30877,  1), (4, 89148,  1),
    (4, 75466,  2), (5, 89148,  2), (5, 47470, 10),
    (6, 41419,  2),
]
```

Por ejemplo, en la venta con boleta número 2, fueron vendidas 4 paltas, 2 bebidas y 6 yogures.

Escriba las siguienes funciones:

```
>>> producto_mas_caro(productos)
```

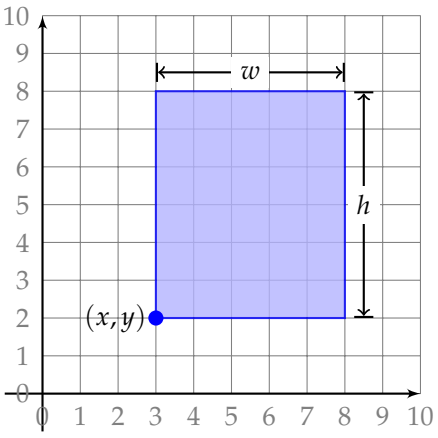


Figura 28.1: El rectángulo `(3, 2, 5, 6)`.

```
'Vuvuzela'
>>> valor_total_bodega(productos)
1900570
>>> ingreso_total_por_ventas(itemes, productos)
13944
>>> producto_con_mas_ingresos(itemes, productos)
'Arroz'
>>> cliente_que_mas_pago(itemes, productos, clientes)
'Fulanita de Tal'
>>> total_ventas_del_mes(2010, 10, itemes, productos)
4160
>>> fecha_ultima_venta_producto(47470, itemes, ventas)
(2010, 10, 13)
```

28.5 Traslape de rectángulos

Un rectángulo que está en el plano xy y cuyos lados son paralelos a los ejes cartesianos puede ser representado mediante cuatro datos: las coordenadas x e y de su vértice inferior izquierdo, su ancho w y su altura h.

En un programa en Python, esto se traduce a una tupla `(x, y, w, h)` de cuatro elementos. Por ejemplo, vea el rectángulo de la figura 28.1.

1. Escriba la función `ingresar_rectangulo()` que pida al usuario ingresar los datos de un rectángulo, y retorne la tupla con los datos ingresados. La función no tiene parámetros. Al ejecutar la función, la sesión debe verse así:

```
Ingrese x: 3
Ingrese y: 2
```

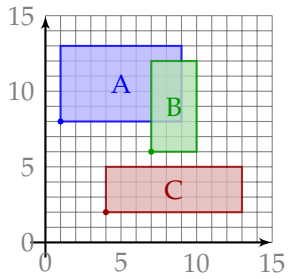


Figura 28.2: Los rectángulos A y B se traslapan. Los rectángulos A y C no se traslapan.

```
Ingrese ancho: 5
Ingrese alto: 6
```

Con esta entrada, la función retornaría la tupla `(3, 2, 5, 6)`.

2. Escriba la función `se_traslapan(r1, r2)` que reciba como parámetros dos rectángulos `r1` y `r2`, y entregue como resultado si los rectángulos se traslapan o no.

   Por ejemplo, con los rectángulos de la figura 28.2:

```
>>> a = (1, 8, 8, 5)
>>> b = (7, 6, 3, 6)
>>> c = (4, 2, 9, 3)
>>> se_traslapan(a, b)
True
>>> se_traslapan(b, c)
False
>>> se_traslapan(a, c)
False
```

3. Escriba un programa que pida al usuario ingresar varios rectángulos, y termine cuando se ingrese uno que se traslape con alguno de los ingresados anteriormente. La salida debe indicar cuáles son los rectángulos que se traslapan.

```
Rectangulo 0
Ingrese x: 4
Ingrese y: 2
Ingrese ancho: 9
Ingrese alto: 3

Rectangulo 1
Ingrese x: 1
Ingrese y: 8
```

```
Ingrese ancho: 8
Ingrese alto: 5

Rectangulo 2
Ingrese x: 11
Ingrese y: 7
Ingrese ancho: 1
Ingrese alto: 9

Rectangulo 3
Ingrese x: 2
Ingrese y: 6
Ingrese ancho: 7
Ingrese alto: 1

Rectangulo 4
Ingrese x: 7
Ingrese y: 6
Ingrese ancho: 3
Ingrese alto: 6
El rectangulo 4 se traslapa con el rectangulo 1
El rectangulo 4 se traslapa con el rectangulo 3
```

28.6 Intersección de circunferencias

En el plano, una circunferencia está determinada por su centro (x, y) y por su radio r. En un programa, podemos representarla como una tupla `(centro, radio)`, donde a su vez `centro` es una tupla `(x, y)`:

```
c = ((4.0, 5.1), 2.8)
```

1. Escriba la función `distancia(p1, p2)` que entregue la distancia entre los puntos `p1` y `p2`:

```
>>> distancia((2, 2), (7, 14))
13.0
>>> distancia((2, 5), (1, 9))
4.1231056256176606
```

2. Escriba la función `se_intersectan(c1, c2)` que indique si las circunferencias `c1` y `c2` se intersectan. Por ejemplo, con las circunferencias de la figura 28.3:

```
>>> A = (( 5.0,  4.0), 3.0)
>>> B = (( 8.0,  6.0), 2.0)
>>> C = (( 8.4, 12.7), 3.0)
>>> D = (( 8.0, 12.0), 2.0)
```

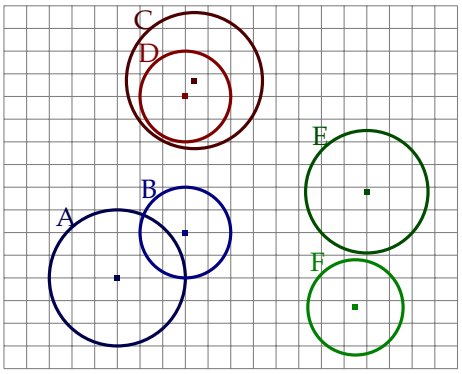


Figura 28.3: Algunos casos de intersecciones entre pares de circunferencias.

```
>>> E = ((16.0,  7.8), 2.7)
>>> F = ((15.5,  2.7), 2.1)
>>> se_intersectan(A, B)
True
>>> se_intersectan(C, D)
False
>>> se_intersectan(E, F)
False
```

Capítulo 29

Listas

29.1 Expresiones con listas

Considere las siguientes listas:

```
>>> a = [5, 1, 4, 9, 0]
>>> b = range(3, 10) + range(20, 23)
>>> c = [[1, 2], [3, 4, 5], [6, 7]]
>>> d = ['perro', 'gato', 'jirafa', 'elefante']
>>> e = ['a', a, 2 * a]
```

Sin usar el computador, indique cuál es el resultado y el tipo de las siguientes expresiones. A continuación, verifique sus respuestas en el computador.

- `a[2]`
- `b[9]`
- `c[1][2]`
- `e[0] == e[1]`
- `len(c)`
- `len(c[0])`
- `len(e)`
- `c[-1]`
- `c[-1][+1]`
- `c[2:] + d[2:]`
- `a[3:10]`
- `a[3:10:2]`

- `d.index('jirafa')`
- `e[c[0][1]].count(5)`
- `sorted(a)[2]`
- `complex(b[0], b[1])`

29.2 Mayores que

Escriba la función `mayores_que(x, valores)` que cuente cuántos valores en la lista `valores` son mayores que `x`:

```
>>> mayores_que(5, [7, 3, 6, 0, 4, 5, 10])
3
>>> mayores_que(2, [-1, 1, 8, 2, 0])
1
```

29.3 Desviación estándar

Desarrolle una función llamada `desviacion_estandar(valores)` cuyo parámetro `valores` sea una lista de números reales, que retorne la *desviación estándar* de éstos, definida como:

$$\sigma = \sqrt{\sum_i \frac{(x_i - \bar{x})^2}{n-1}}$$

donde n es la cantidad de valores, $\bar{x}$ es el promedio de los valores, y los x_i son cada uno de los valores.

Esto significa que hay que hacerlo siguiendo estos pasos:

- calcular el promedio de los valores;
- a cada valor hay que restarle el promedio, y el resultado elevarlo al cuadrado;
- sumar todos los valores obtenidos;
- dividir la suma por la cantidad de valores; y
- sacar la raíz cuadrada del resultado.

```
>>> desviacion_estandar([1.3, 1.3, 1.3])
0.0
>>> desviacion_estandar([4.0, 1.0, 11.0, 13.0, 2.0, 7.0])
4.88535225615
>>> desviacion_estandar([1.5, 9.5])
5.65685424949
```

29.4 Mayores que el promedio

Escriba un programa que pregunte al usuario cuántos datos ingresará, a continuación le pida que ingrese los datos uno por uno, y finalmente entregue como salida cuántos de los datos ingresados son mayores que el promedio.

```
Cuantos datos ingresara? 5
Dato 1: 6.5
Dato 2: 2.1
Dato 3: 2.0
Dato 4: 2.2
Dato 5: 6.1
2 datos son mayores que el promedio
```

```
Cuantos datos ingresara? 10
Dato 1: 9.8
Dato 2: 9.8
Dato 3: 9.8
Dato 4: 9.8
Dato 5: 1.1
Dato 6: 9.8
Dato 7: 9.8
Dato 8: 9.8
Dato 9: 9.8
Dato 10: 9.8
9 datos son mayores que el promedio
```

29.5 Estadísticos de localización

1. La *media aritmética* (o promedio) de un conjunto de datos es la suma de los valores dividida por la cantidad de datos.

   Escriba la función `media_aritmetica(datos)`, donde `datos` es una lista de números, que entregue la media aritmética de los datos:

   ```
   >>> media_aritmetica([6, 1, 4, 8])
   4.75
   ```

2. La *media armónica* de un conjunto de datos es el recíproco de la suma de los recíprocos de los datos, multiplicada por la cantidad de datos:

   $$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n} +}$$

   Escriba la función `media_armonica(datos)`, que entregue la media armónica de los datos:

```
>>> media_armonica([6, 1, 4, 8])
2.5945945945945943
```

3. La *mediana* de un conjunto de datos reales es el valor para el que el conjunto tiene tantos datos mayores como menores a él.

   Más rigurosamente, la mediana es definida de la siguiente manera:

- si la cantidad de datos es impar, la mediana es el valor que queda en la mitad al ordenar los datos de menor a mayor;
- si la cantidad de datos es par, la mediana es el promedio de los dos valores que quedan al centro al ordenar los datos de menor a mayor.

   Escriba la función `mediana(datos)`, que entregue la mediana de los datos:

```
>>> mediana([5.0, 1.4, 3.2])
3.2
>>> mediana([5.0, 1.4, 3.2, 0.1])
2.3
```

   La función no debe modificar la lista que recibe como argumento:

```
>>> x = [5.0, 1.4, 3.2]
>>> mediana(x)
3.2
>>> x
[5.0, 1.4, 3.2]
```

4. La *moda* de un conjunto de datos es el valor que más se repite.

   Escriba la función `modas(datos)`, donde `datos` es una lista, que entregue una lista con las modas de los datos:

```
>>> modas([5, 4, 1, 4, 3, 3, 4, 5, 0])
[4]
>>> modas([5, 4, 1, 4, 3, 3, 4, 5, 3])
[3, 4]
>>> modas([5, 4, 5, 4, 3, 3, 4, 5, 3])
[3, 4, 5]
```

5. Usando las funciones definidas anteriormente, escriba un programa que:

- pregunte al usuario cuántos datos ingresará,
- le pida que ingrese los datos uno por uno, y
- muestre un reporte con las medias aritmética y armónica, la mediana y las modas de los datos ingresados.

   Si alguno de los datos es cero, el reporte no debe mostrar la media armónica.

29.6 Polinomios

Un *polinomio* de grado n es una función matemática de la forma:

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n,$$

donde x es el parámetro y $a_0, a_1, \ldots, a_n$ son números reales dados.

Algunos ejemplos de polinomios son:

- $p(x) = 1 + 2x + x^2$,
- $q(x) = 4 - 17x$,
- $r(x) = -1 - 5x^3 + 3x^5$,
- $s(x) = 5x^{40} + 2x^{80}$.

Los grados de estos polinomios son, respectivamente, 2, 1, 5 y 80.

Evaluar un polinomio significa reemplazar x por un valor y obtener el resultado. Por ejemplo, si evaluamos el polinomio p en el valor $x = 3$, obtenemos el resultado:

$$p(3) = 1 + 2 \cdot 3 + 3^2 = 16.$$

Un polinomio puede ser representado como una lista con los valores $a_0, a_1, \ldots, a_n$. Por ejemplo, los polinomios anteriores pueden ser representados así en un programa:

```
>>> p = [1, 2, 1]
>>> q = [4, -17]
>>> r = [-1, 0, 0, -5, 0, 3]
>>> s = [0] * 40 + [5] + [0] * 39 + [2]
```

1. Escriba la función `grado(p)` que entregue el grado de un polinomio:

```
>>> grado(r)
5
>>> grado(s)
80
```

2. Escriba la función `evaluar(p, x)` que evalúe el polinomio `p` (representado como una lista) en el valor `x`:

```
>>> evaluar(p, 3)
16
>>> evaluar(q, 0.0)
4.0
>>> evaluar(r, 1.1)
-2.82347
>>> evaluar([4, 3, 1], 3.14)
23.2796
```

3. Escriba la función `sumar_polinomios(p1, p2)` que entregue la suma de dos polinomios:

```
>>> sumar_polinomios(p, r)
[0, 2, 1, -5, 0, 3]
```

4. Escriba la función `derivar_polinomio(p)` que entregue la derivada del polinomio `p`:

```
>>> derivar_polinomio(r)
[0, 0, -15, 0, 15]
```

5. Escriba la función `multiplicar_polinomios(p1, p2)` que entregue el producto de dos polinomios:

```
>>> multiplicar_polinomios(p, q)
[4, -9, -30, -17]
```

29.7 Mapear y filtrar

Escriba la función `mapear(f, valores)` cuyos parámetros sean una función `f` y una lista `valores`, y que retorne una nueva lista que tenga los elementos obtenidos al aplicar la función a los elementos de la lista:

```
>>> def cuadrado(x):
...     return x ** 2
...
>>> mapear(cuadrado, [5, 2, 9])
[25, 4, 81]
```

A continuación, escriba la función `filtrar(f, valores)` cuyos parámetros sean una función `f` que retorne un valor booleano y una lista `valores`, y que retorne una nueva lista que tenga todos los elementos de `valores` para los que la función `f` retorna **`True`**:

```
>>> def es_larga(palabra):
...     return len(palabra) > 4
...
>>> p = ['arroz', 'leon', 'oso', 'mochila']
>>> filtrar(es_larga, p)
['arroz', 'mochila']
```

Las funciones no deben modificar la lista original, sino retornar una nueva:

```
>>> filtrar(es_larga, p)
['arroz', 'mochila']
>>> p
['arroz', 'leon', 'oso', 'mochila']
```

(En Python, estas funciones ya existen, y se llaman **`map`** y **`filter`**. Haga como que no lo sabe y escriba las funciones por su cuenta).

29.8 Producto interno

El *producto interno* de dos listas de números es la suma de los productos de los términos correspondientes de ambas. Por ejemplo, si:

```
a = [5, 1, 6]
b = [1, -2, 8]
```

entonces el producto interno entre `a` y `b` es:

```
(5 * 1) + (1 * -2) + (6 * 8)
```

1. Escriba la función `producto_interno(a, b)` que entregue el producto interno de `a` y `b`:

```
>>> a = [7, 1, 4, 9, 8]
>>> b = range(5)
>>> producto_interno(a, b)
68
```

2. Dos listas de números son *ortogonales* si su producto interno es cero. Escriba la función `son_ortogonales(a, b)` que indique si `a` y `b` son ortogonales:

```
>>> son_ortogonales([2, 1], [-3, 6])
True
```

29.9 Ordenamiento

El método `sort` de las listas ordena sus elementos de menor a mayor:

```
>>> a.sort()
>>> a
[0, 1, 4, 6, 9]
```

A veces necesitamos ordenar los elementos de acuerdo a otro criterio. Para esto, el método `sort` acepta un parámetro con nombre llamado `key`, que debe ser una función que asocia a cada elemento el valor que será usado para ordenar. Por ejemplo, para ordenar la lista de mayor a menor uno puede usar una función que cambie el signo de cada número:

```
>>> def negativo(x):
...     return -x
...
>>> a = [6, 1, 4, 0, 9]
>>> a.sort(key=negativo)
>>> a
[9, 6, 4, 1, 0]
```

Esto significa que la lista es ordenada comparando los negativos de sus elementos, aunque son los elementos originales los que aparecen en el resultado.

Como segundo ejemplo, veamos cómo ordenar una lista de números por su último dígito, de menor a mayor:

```
>>> def ultimo_digito(n):
...     return n % 10
...
>>> a = [65, 71, 39, 30, 26]
>>> a.sort(key=ultimo_digito)
>>> a
[30, 71, 65, 26, 39]
```

Resuelva los siguientes problemas de ordenamiento, escribiendo la función criterio para cada caso, e indicando qué es lo que debe ir en la línea marcada con `«???????»`.

- Ordenar una lista de strings de la más corta a la más larga:

```
>>> animales
['conejo', 'ornitorrinco', 'pez', 'hipopotamo', 'tigre']
>>> # ???????
>>> animales
['pez', 'tigre', 'conejo', 'hipopotamo', 'ornitorrinco']
```

- Ordenar una lista de strings de la más larga a la más corta:

```
>>> animales
['conejo', 'ornitorrinco', 'pez', 'hipopotamo', 'tigre']
>>> # ???????
>>> animales
['ornitorrinco', 'hipopotamo', 'conejo', 'tigre', 'pez']
```

- Ordenar una lista de listas según la suma de sus elementos, de menor a mayor:

```
>>> a = [
...   [6, 1, 5, 9],
...   [0, 0, 4, 0, 1],
...   [3, 2, 12, 1],
...   [1000],
...   [7, 6, 1, 0],
... ]
>>> # ??????
>>>
>>> a
[[0, 0, 4, 0, 1], [7, 6, 1, 0], [3, 2, 12, 1],
[6, 1, 5, 9], [1000]]
```

(Las sumas en la lista ordenada son, respectivamente, 5, 14, 18, 21 y 1000).

- Ordenar una lista de tuplas `(nombre, apellido, (anno, mes, dia))` por orden alfabético de apellidos:

```
>>> personas = [
...     ('John',   'Doe',         (1992, 12, 28)),
...     ('Perico', 'Los Palotes', (1992, 10, 8)),
...     ('Yayita', 'Vinagre',     (1991,  4, 17)),
...     ('Fulano', 'De Tal',      (1992, 10, 4)),
... ]
>>> # ???????
>>> from pprint import pprint
>>> pprint(personas)
[('Fulano', 'De Tal', (1992, 10, 4)),
 ('John', 'Doe', (1992, 12, 28)),
 ('Perico', 'Los Palotes', (1992, 10, 8)),
 ('Yayita', 'Vinagre', (1991, 4, 17))]
```

  (La función `pprint` sirve para imprimir estructuras de datos hacia abajo en vez de hacia el lado).

- Ordenar una lista de tuplas `(nombre, apellido, (anno, mes, dia))` por fecha de nacimiento, de la más antigua a la más reciente:

```
>>> # ???????
>>> pprint(personas)
[('Yayita', 'Vinagre', (1991, 4, 17)),
 ('Fulano', 'De Tal', (1992, 10, 4)),
 ('Perico', 'Los Palotes', (1992, 10, 8)),
 ('John', 'Doe', (1992, 12, 28))]
```

- Ordenar una lista de tuplas `(nombre, apellido, (anno, mes, dia))` por fecha de nacimiento, pero ahora de la más reciente a la más antigua:

```
>>> # ???????
>>> pprint(personas)
[('John', 'Doe', (1992, 12, 28)),
 ('Perico', 'Los Palotes', (1992, 10, 8)),
 ('Fulano', 'De Tal', (1992, 10, 4)),
 ('Yayita', 'Vinagre', (1991, 4, 17))]
```

- Ordenar una lista de meses según la cantidad de días, de más a menos:

```
>>> meses = ['agosto', 'noviembre', 'abril', 'febrero']
>>> # ???????
>>> meses
['febrero', 'noviembre', 'abril', 'agosto']
```

- Hacer que queden los números impares a la izquierda y los pares a la derecha:

```
>>> from random import randrange
>>> valores = []
>>> for i in range(12):
...     valores.append(randrange(256))
...
>>> valores
[55, 222, 47, 81, 82, 44, 218, 82, 20, 96, 82, 251]
>>> # ???????
>>> valores
[55, 47, 81, 251, 222, 82, 44, 218, 82, 20, 96, 82]
```

- Hacer que queden los palíndromos a la derecha y los no palíndromos a la izquierda:

```
>>> a = [12321, 584, 713317, 8990, 44444, 28902]
>>> # ????????
>>> a
[584, 8990, 28902, 12321, 713317, 44444]
```

29.10 Iguales o distintos

Escriba la función `todos_iguales(lista)` que indique si todos los elementos de una lista son iguales:

```
>>> todos_iguales([6, 6, 6])
True
>>> todos_iguales([6, 6, 1])
False
>>> todos_iguales([0, 90, 1])
False
```

A continuación, escriba una función `todos_distintos(lista)` que indique si todos los elementos de una lista son distintos:

```
>>> todos_distintos([6, 6, 6])
False
>>> todos_distintos([6, 6, 1])
False
>>> todos_distintos([0, 90, 1])
True
```

Sus funciones deben ser capaces de aceptar listas de cualquier tamaño y con cualquier tipo de datos:

```
>>> todos_iguales([7, 7, 7, 7, 7, 7, 7, 7, 7])
True
>>> todos_distintos(list(range(1000)))
True
```

```
>>> todos_iguales([12])
True
>>> todos_distintos(list('hiperblanduzcos'))
True
```

29.11 Torneo de tenis

Escriba un programa para simular un campeonato de tenis.

Primero, debe pedir al usuario que ingrese los nombres de ocho tenistas. A continuación, debe pedir los resultados de los partidos juntando los jugadores de dos en dos. El ganador de cada partido avanza a la ronda siguiente.

El programa debe continuar preguntando ganadores de partidos hasta que quede un único jugador, que es el campeón del torneo.

El programa en ejecución debe verse así:

```
Jugador 1: Nadal
Jugador 2: Melzer
Jugador 3: Murray
Jugador 4: Soderling
Jugador 5: Djokovic
Jugador 6: Berdych
Jugador 7: Federer
Jugador 8: Ferrer

Ronda 1
a.Nadal - b.Melzer: a
a.Murray - b.Soderling: b
a.Djokovic - b.Berdych: a
a.Federer - b.Ferrer: a

Ronda 2
a.Nadal - b.Soderling: a
a.Djokovic - b.Federer: a

Ronda 3
a.Nadal - b.Djokovic: b

Campeon: Djokovic
```

Capítulo 30

Conjuntos y diccionarios

30.1 Expresiones con conjuntos

Considere las siguientes asignaciones:

```
>>> a = {5, 2, 3, 9, 4}
>>> b = {3, 1}
>>> c = {7, 5, 5, 1, 8, 6}
>>> d = [6, 2, 4, 5, 5, 3, 1, 3, 7, 8]
>>> e = {(2, 3), (3, 4), (4, 5)}
>>> f = [{2, 3}, {3, 4}, {4, 5}]
```

Sin usar el computador, indique cuál es el resultado y el tipo de las siguientes expresiones. A continuación, verifique sus respuestas en el computador.

- `len(c)`
- `len(set(d))`
- `a & (b | c)`
- `(a & b) | c`
- `c - a`
- `max(e)`
- `f[0] < a`
- `set(range(4)) & a`
- `(set(range(4)) & a) in f`
- `len(set('perro'))`
- `len({'perro'})`

30.2 Conjunto potencia

El *conjunto potencia* de un conjunto S es el conjunto de todos los subconjuntos de S. Por ejemplo, el conjunto potencia de $\{1,2,3\}$ es:

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$$

En Python, un conjunto no puede contener a otros conjuntos, ya que no puede tener elementos mutables, y los conjuntos lo son:

```
>>> a = set()
>>> a.add({1, 2})        # :(
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Lo que sí podemos crear es una lista de conjuntos:

```
>>> l = list()
>>> l.append({1, 2})     # :)
>>> l
[set([1, 2])]
```

Escriba la función `conjunto_potencia(s)` que reciba como parámetro un conjunto cualquiera `s` y retorne su «lista potencia» (la lista de todos sus subconjuntos):

```
>>> conjunto_potencia({6, 1, 4})
[set(), set([6]), set([1]), set([4]), set([6, 1]),
set([6, 4]), set([1, 4]), set([6, 1, 4])]
```

30.3 Expresiones con diccionarios

Considere las siguientes asignaciones:

```
>>> a = {'a': 14, 'b': 23, 'c': 88}
>>> b = {12: True, 55: False, -2: False}
>>> c = dict()
>>> d = {1: [2, 3, 4], 5: [6, 7, 8, 9], 10: [11]}
>>> e = {2 + 3: 4, 5: 6 + 7, 8: 9, 10: 11 + 12}
```

Sin usar el computador, indique cuál es el resultado y el tipo de las siguientes expresiones. A continuación, verifique sus respuestas en el computador.

- `a['c']`
- `a[23]`
- `b[-2] or b[55]`
- `23 in a`

- `'a' in a`
- `5 in d[5]`
- `sum(b)`
- `len(c)`
- `len(d)`
- `len(d[1])`
- `len(b.values())`
- `len(e)`
- `sum(a.values())`
- `max(list(e))`
- `d[1] + d[5] + d[10]`
- `max(map(len, d.values()))`

30.4 Contar letras y palabras

1. Escriba la función `contar_letras(oracion)` que retorne un diccionario asociando a cada letra la cantidad de veces que aparece en la oracion:

   ```
   >>> contar_letras('El elefante avanza hacia Asia')
   {'a': 8, 'c': 1, 'e': 4, 'f': 1, 'h': 1, 'i': 2, 'l': 2,
   'n': 2, 's': 1, 't': 1, 'v': 1, 'z': 1}
   ```

   Cada valor del diccionario debe considerar tanto las apariciones en mayúscula como en minúscula de la letra correspondiente. Los espacios deben ser ignorados.

2. Escriba la función `contar_vocales(oracion)` que retorne un diccionario asociando a cada vocal la cantidad de veces que aparece en la oracion. Si una vocal no aparece en la oración, de todos modos debe estar en el diccionario asociada al valor 0:

   ```
   >>> contar_vocales('El elefante avanza hacia Asia')
   {'a': 8, 'e': 4, 'i': 2, 'o': 0, 'u': 0}
   ```

3. Escriba la función `contar_iniciales(oracion)` que retorne un diccionario asociando a cada letra la cantidad de veces que aparece al principio de una palabra:

```
>>> contar_iniciales('El elefante avanza hacia Asia')
{'e': 2, 'h': 1, 'a': 2}
>>> contar_iniciales('Varias vacas vuelan sobre Venezuela')
{'s': 1', 'v': 4}
```

4. Escriba la función `obtener_largo_palabras(oracion)` que retorne un diccionario asociando a cada palabra su cantidad de letras:

   ```
   >>> obtener_largo_palabras('el gato y el pato son amigos')
   {'el': 2, 'son': 3, 'gato': 4, 'y': 1, 'amigos': 6, 'pato': 4}
   ```

5. Escriba la función `contar_palabras(oracion)` que retorne un diccionario asociando a cada palabra la cantidad de veces que aparece en la oración:

   ```
   >>> contar_palabras('El sobre esta sobre el pupitre')
   {'sobre': 2, 'pupitre': 1, 'el': 2, 'esta': 1}
   ```

6. Escriba la función `palabras_repetidas(oracion)` que retorne una lista con las palabras que están repetidas:

   ```
   >>> palabras_repetidas('El partido termino cero a cero')
   ['cero']
   >>> palabras_repetidas('El sobre esta sobre el mueble')
   ['el', 'sobre']
   >>> palabras_repetidas('Ay, ahi no hay pan')
   []
   ```

   Para obtener la lista de palabras de la oración, puede usar el método `split` de los strings:

   ```
   >>> s = 'el gato y el pato'
   >>> s.split()
   ['el', 'gato', 'y', 'el', 'pato']
   ```

   Para obtener un string en minúsculas, puede usar el método `lower`:

   ```
   >>> s = 'Venezuela'
   >>> s.lower()
   'venezuela'
   ```

30.5 Recorrido de diccionarios

1. Escriba la función `hay_llaves_pares(d)` que indique si el diccionario `d` tiene alguna llave que sea un número par.

   A continuación, escriba una función `hay_valores_pares(d)` que indique si el diccionario `d` tiene algún valor que sea un número par.

   Para probar las funciones, ocupe diccionarios cuyas llaves y valores sean sólo números enteros:

```
>>> d1 = {1: 2, 3: 5}
>>> d2 = {2: 1, 6: 7}
>>> hay_valores_pares(d1)
True
>>> hay_valores_pares(d2)
False
>>> hay_llaves_pares(d1)
False
>>> hay_llaves_pares(d2)
True
```

2. Escriba la función `maximo_par(d)` que entregue el valor máximo de la suma de una llave y un valor del diccionario `d`:

```
>>> d = {5: 1, 4: 7, 9: 0, 2: 2}
>>> maximo_par(d)
11
```

3. Escriba la función `invertir(d)` que entregue un diccionario cuyas llaves sean los valores de `d` y cuyos valores sean las llaves respectivas:

```
>>> invertir({1: 2, 3: 4, 5: 6})
{2: 1, 4: 3, 6: 5}
>>> apodos = {
...   'Suazo': 'Chupete',
...   'Sanchez': 'Maravilla',
...   'Medel': 'Pitbull',
...   'Valdivia': 'Mago',
... }
>>> invertir(apodos)
{'Maravilla': 'Sanchez', 'Mago': 'Valdivia',
'Chupete': 'Suazo', 'Pitbull': 'Medel'}
```

Capítulo 31

Uso de estructuras de datos

31.1 Expresiones con estructuras de datos anidadas

Considere el siguiente trozo de programa:

```
d = {
  (1, 2): [{1, 2}, {3}, {1, 3}],
  (2, 1): [{3}, {1, 2}, {1, 2, 3}],
  (2, 2): [{}, {2, 3}, {1, 3}],
}
```

Indique el valor y el tipo de las siguientes expresiones:

- `len(d)` (respuesta: el valor es `3`, el tipo es `int`).
- `d[(1, 2)][2]` (respuesta: el valor es `{1, 3}`, el tipo es `set`).
- `d[(2, 2)][0]`
- `(1, 2)`
- `(1, 2)[1]`
- `d[(1, 2)][1]`
- `d[(1, 2)]`
- `d[1, 2]`
- `len(d[2, 1])`
- `len(d[2, 1][1])`
- `d[2, 2][1] & d[1, 2][2]`
- `(d[2, 2] + d[2, 1])[4]`
- `max(map(len, d.values()))`

- `min(map(len, d[2, 1]))`
- `d[1, 2][-3] & d[2, 1][-2] & d[2, 2][-1]`
- `d[len(d[2, 1][1]), len(d[1, 2][-1])][1]`

Puede verificar sus respuestas usando la consola interactiva. Para obtener el tipo, use la función `type`:

```
>>> v = d[(1, 2)][2]
>>> v
{1, 3}
>>> type(v)
<class 'set'>
```

31.2 Países

El diccionario `paises` asocia cada persona con el conjunto de los países que ha visitado:

```
paises = {
    'Pepito': {'Chile', 'Argentina'},
    'Yayita': {'Francia', 'Suiza', 'Chile'},
    'John': {'Chile', 'Italia', 'Francia', 'Peru'},
}
```

Escriba la funcion `cuantos_en_comun(a, b)`, que retorne la cantidad de países en común que han visitado la persona `a` y la persona `b`:

```
>>> cuantos_en_comun('Pepito', 'John')
1
>>> cuantos_en_comun('John', 'Yayita')
2
```

31.3 Signo zodiacal

El signo zodiacal de una persona está determinado por su día de nacimiento. El diccionario `signos` asocia a cada signo el período del año que le corresponde. Cada período es una tupla con la fecha de inicio y la fecha de término, y cada fecha es una tupla `(mes, dia)`:

```
signos = {
   'aries':       (( 3, 21), ( 4, 20)),
   'tauro':       (( 4, 21), ( 5, 21)),
   'geminis':     (( 5, 22), ( 6, 21)),
   'cancer':      (( 6, 22), ( 7, 23)),
   'leo':         (( 7, 24), ( 8, 23)),
   'virgo':       (( 8, 24), ( 9, 23)),
```

```
    'libra':       (( 9, 24), (10, 23)),
    'escorpio':    ((10, 24), (11, 22)),
    'sagitario':   ((11, 23), (12, 21)),
    'capricornio': ((12, 22), ( 1, 20)),
    'acuario':     (( 1, 21), ( 2, 19)),
    'piscis':      (( 2, 20), ( 3, 20)),
}
```

Por ejemplo, para que una persona sea de signo libra debe haber nacido entre el 24 de septiembre y el 23 de octubre.

Escriba la función `determinar_signo(fecha_de_nacimiento)` que reciba como parámetro la fecha de nacimiento de una persona, representada como una tupla `(anno, mes, dia)`, y que retorne el signo zodiacal de la persona:

```
>>> determinar_signo((1990, 5, 7))
'tauro'
>>> determinar_signo((1904, 11, 24))
'sagitario'
>>> determinar_signo((1998, 12, 28))
'capricornio'
>>> determinar_signo((1999, 1, 11))
'capricornio'
```

31.4 Asistencia

La asistencia de los alumnos a clases puede ser llevada en una tabla como la siguiente:

Clase	1	2	3	4	5	6	7
Pepito	✓	✓	✓				
Yayita	✓	✓	✓		✓		✓
Fulanita	✓	✓	✓	✓	✓	✓	✓
Panchito	✓	✓	✓		✓	✓	✓

En un programa, esta informacion puede ser representada usando listas:

```
>>> alumnos = ['Pepito', 'Yayita', 'Fulanita', 'Panchito']
>>> asistencia = [
...  [True, True, True, False, False, False, False],
...  [True, True, True, False, True,  False, True ],
...  [True, True, True, True,  True,  True,  True ],
...  [True, True, True, False, True,  True,  True ]]
>>>
```

1. Escriba la función `total_por_alumno(tabla)` que reciba como parámetro la tabla de asistencia y retorne una lista con el número de clases a las que asistió cada alumno:

```
>>> total_por_alumno(asistencia)
[3, 5, 7, 6]
```

2. Escriba la función `total_por_clase(tabla)` que reciba como parámetro la tabla de asistencia y retorne una lista con el número de alumnos que asistió a cada clase:

```
>>> total_por_clase(asistencia)
[4, 4, 4, 1, 3, 2, 3]
```

3. Escriba la función `alumno_estrella(asistencia)` que indique qué alumno asistió más a clases:

```
>>> alumno_estrella(asistencia)
'Fulanita'
```

31.5 Cumpleaños

Las fechas pueden ser representadas como tuplas `(ano, mes, dia)`. Para asociar a cada persona su fecha de nacimiento, se puede usar un diccionario:

```
>>> n = {
...     'Pepito': (1990, 10, 20),
...     'Yayita': (1992, 3, 3),
...     'Panchito': (1989, 10, 20),
...     'Perica': (1989, 12, 8),
...     'Fulanita': (1991, 2, 14),
... }
```

1. Escriba la función `mismo_dia(fecha1, fecha2)` que indique si las dos fechas ocurren el mismo día del año (aunque sea en años diferentes):

```
>>> mismo_dia((2010, 6, 11), (1990, 6, 11))
True
>>> mismo_dia((1981, 8, 12), (1981, 5, 12))
False
```

2. Escriba la función `mas_viejo(n)` que indique quién es la persona más vieja según las fechas de nacimiento del diccionario `n`:

```
>>> mas_viejo(n)
'Panchito'
```

3. Escriba la función `primer_cumple(n)` que indique quién es la persona que tiene el primer cumpleaños del año:

```
>>> primer_cumple(n)
'Fulanita'
```

31.6 Conjugador de verbos

Escriba un programa que reciba como entrada el infinitivo de un verbo regular y a continuación muestre su conjugación en tiempo presente:

```
Ingrese verbo: amar
yo amo
tu amas
el ama
nosotros amamos
vosotros amais
ellos aman
```

```
Ingrese verbo: comer
yo como
tu comes
el come
nosotros comemos
vosotros comeis
ellos comen
```

```
Ingrese verbo: vivir
yo vivo
tu vives
el vive
nosotros vivimos
vosotros vivis
ellos viven
```

Utilice un diccionario para asociar a cada terminación (-ar, -er e -ir) sus declinaciones, y una lista para guardar los pronombres en orden:

```
pronombres = ['yo', 'tu', 'el',
              'nosotros', 'vosotros', 'ellos']
```

31.7 Acordes

En teoría musical, la escala cromática está formada por doce notas:

```
notas = ['do', 'do#', 're', 're#', 'mi', 'fa',
         'fa#', 'sol', 'sol#', 'la', 'la#', 'si']
```

(El signo ♯ se lee «sostenido»). Cada nota corresponde a una tecla del piano, como se muestra en la figura 31.1. Los sostenidos son las teclas negras. La escala es circular, y se extiende infinitamente en ambas direcciones. Esto signfica que después de si viene nuevamente do.

Cada par de notas consecutivas está separada por un semitono. Por ejemplo, entre re y sol♯ hay 6 semitonos.

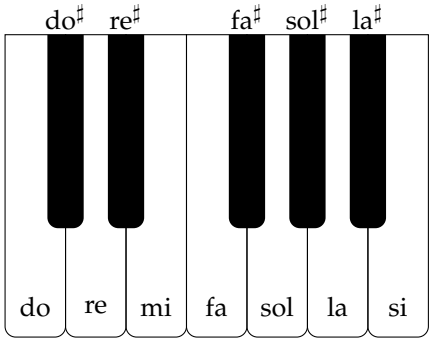


Figura 31.1: Correspondencia entre notas y teclas del piano.

Un *acorde* es una combinación de notas que suenan bien al unísono.

Existen varios tipos de acordes, que difieren en la cantidad de semitonos por las que sus notas están separadas. Por ejemplo, los acordes mayores tienen tres notas separadas por 4 y 3 semitonos. Así es como el acorde de re mayor está formado por las notas:

re, fa♯ y la,

pues entre re y fa♯ hay 4 semitonos, y entre fa♯ y la, 3.

Algunos tipos de acordes están presentados en el siguiente diccionario, asociados a las separaciones entre notas consecutivas del acorde:

```
acordes = {
    'mayor': (4, 3),
    'menor': (3, 4),
    'aumentado': (4, 4),
    'disminuido': (3, 3),
    'sus 2': (2, 5),
    'sus 4': (5, 2),
    '5': (7,),
    'menor 7': (3, 4, 3),
    'mayor 7': (4, 3, 4),
    '7': (4, 3, 3),
}
```

Escriba la función `acorde(nota, tipo)` que entegue una lista de las notas del acorde en el orden correcto:

```
>>> acorde('la', 'mayor')
['la', 'do#', 'mi']
>>> acorde('sol#', 'menor')
['sol#', 'si', 'do#']
>>> acorde('si', '7')
```

```
['si', 're#', 'fa#', 'la']
>>> acorde('do#', '5')
['do#', 'sol#']
```

Si el tipo no es entregado, la función debe suponer que el acorde es mayor:

```
>>> acorde('si')
['si', 're#', 'fa#']
```

31.8 Campeonato de fútbol

Los resultados de un campeonato de fútbol están almacenados en un diccionario. Las llaves son los partidos y los valores son los resultados. Cada partido es representado como una tupla con los dos equipos que jugaron, y el resultado es otra tupla con los goles que hizo cada equipo:

```
resultados = {
    ('Honduras', 'Chile'):    (0, 1),
    ('Espana',   'Suiza'):    (0, 1),
    ('Suiza',    'Chile'):    (0, 1),
    ('Espana',   'Honduras'): (3, 0),
    ('Suiza',    'Honduras'): (0, 0),
    ('Espana',   'Chile'):    (2, 1),
}
```

1. Escriba la función `obtener_lista_equipos(resultados)` que reciba como parámetro el diccionario de resultados y retorne una lista con todos los equipos que participaron del campeonato:

   ```
   >>> obtener_lista_equipos(resultados)
   ['Honduras', 'Suiza', 'Espana', 'Chile']
   ```

2. El equipo que gana un partido recibe tres puntos y el que pierde, cero. En caso de empate, ambos equipos reciben un punto.

   Escriba la función `calcular_puntos(equipo, resultados)` que entregue la cantidad de puntos obtenidos por un equipo:

   ```
   >>> calcular_puntos('Chile', resultados)
   6
   >>> calcular_puntos('Suiza', resultados)
   4
   ```

3. La *diferencia de goles* de un equipo es el total de goles que anotó un equipo menos el total de goles que recibió.

   Escriba la función `calcular_diferencia_de_goles(equipo, resultados)` que entregue la diferencia de goles de un equipo:

```
>>> calcular_diferencia_de_goles('Chile', resultados)
1
>>> calcular_diferencia_de_goles('Honduras', resultados)
-4
```

4. Escriba la función `posiciones(resultados)` que reciba como parámetro el diccionario de resultados, y retorne una lista con los equipos ordenados por puntaje de mayor a menor. Los equipos que tienen el mismo puntaje deben ser ordenados por diferencia de goles de mayor a menor. Si tienen los mismos puntos y la misma diferencia de goles, deben ser ordenados por los goles anotados:

   ```
   >>> posiciones(resultados)
   ['Espana', 'Chile', 'Suiza', 'Honduras']
   ```

   En este ejemplo, España queda clasificado en primer lugar porque tiene 6 puntos y diferencia de goles de $+3$, mientras que Chile también tiene 6 puntos, pero diferencia de goles de $+1$.

31.9 Manos de póker

En los juegos de naipes, una carta tiene dos atributos: un valor (2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A) y un palo (♡, ♢, ♣, ♠).

En un programa, el valor puede ser representado como un número del 1 al 13, y el palo como un string: ♡ → `'C'`, ♢ → `'D'`, ♣ → `'T'` y ♠ → `'P'`.

Una carta puede ser representada como una tupla de dos elementos: el valor y el palo:

```
carta1 = (5, 'T')
carta2 = (10, 'D')
```

Para simplificar, se puede representar el as como un 1, y los «monos» J, Q y K como 11, 12 y 13:

```
carta3 = (1, 'P')    # as de picas
carta4 = (12, 'C')   # reina de corazones
```

En el juego de póker, una mano tiene cinco cartas, lo que en un programa vendría a ser un conjunto de cinco tuplas:

```
mano = {(1, 'P'), (1, 'C'), (1, 'T'), (13, 'D'), (12, 'P')}
```

1. Un *full* es una mano en que tres cartas tienen el mismo valor, y las otras dos tienen otro valor común. Por ejemplo, A♠ A♡ 6♣ A♢ 6♢ es un full (tres ases y dos seis), pero 2♣ A♡ Q♡ A♢ 6♢ no.

   Escriba la función que indique si la mano es un full:

```
>>> a = {(1, 'P'), (1, 'C'), (6, 'T'), (1, 'D'), (6, 'D')}
>>> b = {(2, 'T'), (1, 'C'), (12, 'C'), (1, 'D'), (6, 'D')}
>>> es_full(a)
True
>>> es_full(b)
False
```

2. Un *color* es una mano en que todas las cartas tienen el mismo palo. Por ejemplo, 8♠ K♠ 4♠ 9♠ 2♠ es un color (todas las cartas son picas), pero Q♣ A♡ 5♡ 2♡ 2♢ no lo es.

   Escriba la función que indique si la mano es un color:

```
>>> a = {(8, 'P'), (13, 'P'), (4, 'P'), (9, 'P'), (2, 'P')}
>>> b = {(12, 'T'), (1, 'C'), (5, 'C'), (2, 'C'), (2, 'D')}
>>> es_color(a)
True
>>> es_color(b)
False
```

3. Una *escalera* es una mano en que las cartas tienen valores consecutivos. Por ejemplo, 4♠ 7♡ 3♡ 6♣ 5♣ es una escalera (tiene los valores 3, 4, 5, 6 y 7), pero Q♣ 7♡ 3♡ Q♡ 5♣ no lo es.

   Escriba la función que indique si la mano es una escalera:

```
>>> a = {(4, 'P'), (7, 'C'), (3, 'C'), (6, 'T'), (5, 'T')}
>>> b = {(12, 'T'), (7, 'C'), (3, 'C'), (12, 'C'), (5, 'T')}
>>> es_escalera(a)
True
>>> es_escalera(b)
False
```

4. Una *escalera de color* es una escalera en la que todas las cartas tienen el mismo palo. Por ejemplo, 4♢ 7♢ 3♢ 6♢ 5♢ es una escalera de color (son sólo diamantes, y los valores 3, 4, 5, 6 y 7 son consecutivos).

   Escriba la función que indique si la mano es una escalera de color:

```
>>> a = {(4, 'P'), (7, 'C'), (3, 'C'), (6, 'T'), (5, 'T')}
>>> b = {(8, 'P'), (13, 'P'), (4, 'P'), (9, 'P'), (2, 'P')}
>>> c = {(4, 'D'), (7, 'D'), (3, 'D'), (6, 'D'), (5, 'D')}
>>> es_escalera_de_color(a)
False
>>> es_escalera_de_color(b)
False
>>> es_escalera_de_color(c)
True
```

5. Escriba las funciones para identificar las demás manos del póker.

6. Escriba un programa que pida al usuario ingresar cinco cartas, y le indique qué tipo de mano es:

```
Carta 1: 5D
Carta 2: QT
Carta 3: QD
Carta 4: 10P
Carta 5: 5C
Doble pareja
```

Capítulo 32

Procesamiento de texto

32.1 Telégrafo

Dado un mensaje, se debe calcular su costo para enviarlo por telégrafo. Para esto se sabe que cada letra cuesta $10, los caracteres especiales que no sean letras cuestan $30 y los dígitos tienen un valor de $20 cada uno. Los espacios no tienen valor.

Su mensaje debe ser un string, y las letras del castellano (ñ, á, é, í, ó, ú) se consideran caracteres especiales.

```
Mensaje: Feliz Aniversario!
Su mensaje cuesta $190
```

32.2 Palabras especiales

1. Dos palabras son *anagramas* si tienen las mismas letras pero en otro orden. Por ejemplo, «torpes» y «postre» son anagramas, mientras que «aparta» y «raptar» no lo son, ya que «raptar» tiene una *r* de más y una *a* de menos.

   Escriba la función `son_anagramas(p1, p2)` que indique si las palabras `p1` y `p2` son anagramas:

   ```
   >>> son_anagramas('torpes', 'postre')
   True
   >>> son_anagramas('aparta', 'raptar')
   False
   ```

2. Las palabras *panvocálicas* son las que tienen las cinco vocales. Por ejemplo: centrifugado, bisabuelo, hipotenusa.

   Escriba la función `es_panvocalica(palabra)` que indique si una palabra es panvocálica o no:

   ```
   >>> es_panvocalica('educativo')
   True
   ```

```
>>> es_panvocalica('pedagogico')
False
```

3. Escriba la función `cuenta_panvocalicas(oracion)` que cuente cuántas palabras panvocálicas tiene una oración:

```
>>> cuenta_panvocalicas('la contertulia estudiosa va a casa')
2
>>> cuenta_panvocalicas('los hipopotamos bailan al amanecer')
0
oracion = 'el abuelito mordisquea el aceituno con contundencia'
>>> cuenta_panvocalicas(oracion)
4
```

4. Escriba la función `tiene_letras_en_orden(palabra)` que indique si las letras de la palabra están en orden alfabético:

```
>>> tiene_letras_en_orden('himnos')
True
>>> tiene_letras_en_orden('abenuz')
True
>>> tiene_letras_en_orden('zapato')
False
```

5. Escriba la función `tiene_letras_dos_veces(palabra)` que indique si cada letra de la palabra aparece exactamente dos veces:

```
>>> tiene_letras_dos_veces('aristocraticos')
True
>>> tiene_letras_dos_veces('quisquilloso')
True
>>> tiene_letras_dos_veces('aristocracia')
False
```

6. Escriba la función `palabras_repetidas(oracion)` que entregue una lista de las palabras que están repetidas en la oración:

```
>>> palabras_repetidas('El partido termino cero a cero')
['cero']
>>> palabras_repetidas('El sobre esta sobre el mueble')
['el', 'sobre']
>>> palabras_repetidas('Ay, ahi no hay pan')
[]
```

7. Un *pangrama* es un texto que tiene todas las letras del alfabeto, de la *a* a la *z* (por las limitaciones de Python 2.7 excluiremos la *ñ*). Escriba la función `es_pangrama(texto)` que indique si el texto es o no un pangrama:

```
>>> a = 'Sylvia wagt quick den Jux bei Pforzheim.'
>>> b = 'Cada vez que trabajo, Felix me paga un whisky.'
>>> c = 'Cada vez que trabajo, Luis me invita a una cerveza.'
>>> es_pangrama(a)
True
>>> es_pangrama(b)
True
>>> es_pangrama(c)
False
```

32.3 Vocales y consonantes

Escriba una función que determine si una letra es vocal o consonante. Decida usted qué es lo que retornará la función. Por ejemplo, podría ser así:

```
>>> es_vocal('a')
True
>>> es_vocal('b')
False
```

O así:

```
>>> es_consonante('a')
False
>>> es_consonante('b')
True
```

O incluso así:

```
>>> tipo_de_letra('a')
'vocal'
>>> tipo_de_letra('b')
'consonante'
```

A continuación, escriba una función que retorne las cantidades de vocales y consonantes de la palabra. Esta función debe llamar a la función que usted escribió antes.

```
>>> v, c = contar_vocales_y_consonantes('edificio')
>>> v
5
>>> c
3
```

Finalmente, escriba un programa que pida al usuario ingresar una palabra y le muestre cuántas vocales y consonantes tiene:

```
Ingrese palabra: edificio
Tiene 5 vocales y 3 consonantes
```

32.4 Análisis de correos electrónicos

La empresa RawInput S.A. desea hacer una segmentación de sus clientes según su ubicación geográfica. Para esto, analizará su base de datos de correos electrónicos con el fin obtener información sobre el lugar de procedencia de cada cliente.

En una dirección de correo electrónico, el *dominio* es la parte que va después de la arroba, y el *TLD* es lo que va después del último punto. Por ejemplo, en la dirección `fulano.de.tal@alumnos.usm.cl`, el dominio es `alumnos.usm.cl` y el TLD es `cl`.

Algunos TLD no están asociados a un país, sino que representan otro tipo de entidades. Estos TLD genéricos son los siguentes:

```
genericos = {'com', 'gov', 'edu', 'org', 'net', 'mil'}
```

1. Escriba la función `obtener_dominios(correos)` que reciba como parámetro una lista de correos electrónicos, y retorne la lista de todos los dominios, sin repetir, y en orden alfabético.

2. Escriba la función `contar_tld(correos)` que reciba como parámetro la lista de correos electrónicos, y retorne un diccionario que asocie a cada TLD la cantidad de veces que aparece en la lista. No debe incluir los TLD genéricos.

```
>>> c = ['fulano@usm.cl', 'erika@lala.de', 'li@zi.cn',
...      'a@a.net', 'gudrun@lala.de', 'yayita@abc.cl',
...      'otto.von.d@lorem.ipsum.de', 'org@cn.de.cl',
...      'jozin@baz.cz', 'jp@foo.cl', 'dawei@hao.cn',
...      'pepe@gmail.com', 'ana@usm.cl', 'fer@x.com',
...      'ada@alumnos.usm.cl', 'dj@foo.cl', 'jan@baz.cz',
...      'polo@hotmail.com', 'd@abc.cl']
>>> obtener_dominios(c)
['abc.cl', 'alumnos.usm.cl', 'baz.cz', 'cn.de.cl', 'foo.cl',
'hao.cn', 'lala.de', 'lorem.ipsum.de', 'usm.cl', 'zi.cn']
>>> contar_tld(c)
{'cz': 2, 'de': 3, 'cn': 2, 'cl': 8}
```

Capítulo 33

Archivos de texto

33.1 Sumas por fila y columna

El archivo `datos1.txt` tiene tres números enteros en cada línea:

```
45 12 98
1 12 65
7 15 76
54 23 1
65 2 84
```

1. Escriba la función `suma_lineas(nombre_archivo)` que entregue una lista con las sumas de todas las líneas del archivo:

   ```
   >>> suma_lineas('datos1.txt')
   [155, 78, 98, 78, 151]
   ```

2. Escriba la función `suma_columnas(nombre_archivo)` que entregue una lista con las sumas de las tres columnas del archivo:

   ```
   >>> suma_columnas('datos1.txt')
   [172, 64, 324]
   ```

33.2 Reporte de notas

Las notas de un ramo están almacenadas en un archivo llamado `notas.txt`, que contiene lo siguiente:

```
Pepito:5.3:3.7:6.7:6.7:7.1:5.5
Yayita:5.5:5.2:2.0:5.6:6.0:2.0
Fulanita:7.1:6.6:6.4:5.1:5.8:6.3
Moya:5.2:4.7:1.8:3.5:2.7:4.5
```

Cada línea tiene el nombre del alumno y sus seis notas, separadas por dos puntos («:»).

Escriba un programa que cree un nuevo archivo llamado `reporte.txt`, en que cada línea indique si el alumno está aprobado (promedio $\geq$ 4,0) o reprobado (promedio $<$ 4,0):

```
Pepito aprobado
Yayita aprobado
Fulanita aprobado
Moya reprobado
```

33.3 Consulta médica

Una consulta médica tiene un archivo `pacientes.txt` con los datos personales de sus pacientes. Cada línea del archivo tiene el rut, el nombre y la edad de un paciente, separados por un símbolo «:». Así se ve el archivo:

```
12067539-7:Anastasia Lopez:32
15007265-4:Andres Morales:26
8509454-8:Pablo Munoz:45
7752666-8:Ignacio Navarro:49
8015253-1:Alejandro Pacheco:51
9217890-0:Patricio Pimienta:39
9487280-4:Ignacio Rosas:42
12393241-2:Ignacio Rubio:33
11426761-9:Romina Perez:35
15690109-1:Francisco Ruiz:26
6092377-9:Alfonso San Martin:65
9023365-3:Manuel Toledo:38
10985778-5:Jesus Valdes:38
13314970-8:Abel Vazquez:30
7295601-k:Edison Munoz:60
5106360-0:Andrea Vega:71
8654231-5:Andres Zambrano:55
10105321-0:Antonio Almarza:31
13087677-3:Jorge Alvarez:28
9184011-1:Laura Andrade:47
12028339-1:Jorge Argandona:29
10523653-0:Camila Avaria:40
12187197-1:Felipe Avila:36
5935556-2:Aquiles Barriga:80
14350739-4:Eduardo Bello:29
6951420-0:Cora Benitez:68
11370775-5:Hugo Berger:31
11111756-k:Cristobal Borquez:34
```

Además, cada vez que alguien se atiende en la consulta, la visita es registrada en el archivo `atenciones.txt`, agregando una línea que tiene el rut del paciente, la fecha de la visita (en formato `dia-mes-año`) y el precio de la atención, también separados por «`:`». El archivo se ve así:

```
8015253-1:4-5-2010:69580
12393241-2:6-5-2010:57274
10985778-5:8-5-2010:73206
8015253-1:10-5-2010:30796
8015253-1:12-5-2010:47048
12028339-1:12-5-2010:47927
11426761-9:13-5-2010:39117
10985778-5:15-5-2010:86209
7752666-8:18-5-2010:41916
8015253-1:18-5-2010:74101
12187197-1:20-5-2010:38909
8654231-5:20-5-2010:75018
8654231-5:22-5-2010:64944
5106360-0:24-5-2010:53341
8015253-1:27-5-2010:76047
9217890-0:30-5-2010:57726
7752666-8:1-6-2010:54987
8509454-8:2-6-2010:76483
6092377-9:2-6-2010:62106
11370775-5:3-6-2010:67035
11370775-5:7-6-2010:47299
8509454-8:7-6-2010:73254
8509454-8:10-6-2010:82955
11111756-k:10-6-2010:56520
7752666-8:10-6-2010:40820
12028339-1:12-6-2010:79237
11111756-k:13-6-2010:69094
5935556-2:14-6-2010:73174
11111756-k:21-6-2010:70417
11426761-9:22-6-2010:80217
12067539-7:25-6-2010:31555
11370775-5:26-6-2010:75796
10523653-0:26-6-2010:34585
6951420-0:28-6-2010:45433
5106360-0:1-7-2010:48445
8654231-5:4-7-2010:76458
```

Note que las fechas están ordenadas de menos a más reciente, ya que las nuevas líneas siempre se van agregando al final.

1. Escriba la función `costo_total_paciente(rut)` que entregue el costo total de las atenciones del paciente con el rut dado:

```
>>> costo_total_paciente('8015253-1')
297572
>>> costo_total_paciente('14350739-4')
0
```

2. Escriba la función `pacientes_dia(dia, mes, ano)` que entregue una lista con los nombres de los pacientes que se atendieron el día señalado:

```
>>> pacientes_dia(2, 6, 2010)
['Pablo Munoz', 'Alfonso San Martin']
>>> pacientes_dia(23, 6, 2010)
[]
```

3. Escriba la función `separar_pacientes()` que construya dos nuevos archivos:

- `jovenes.txt`, con los datos de los pacientes menores de 30 años;
- `mayores.txt`, con los datos de todos los pacientes mayores de 60 años.

Por ejemplo, el archivo `jovenes.txt` debe verse así:

```
15007265-4:Andres Morales:26
15690109-1:Francisco Ruiz:26
13087677-3:Jorge Alvarez:28
12028339-1:Jorge Argandona:29
14350739-4:Eduardo Bello:29
```

4. Escribir una función `ganancias_por_mes()` que construya un nuevo archivo llamado `ganancias.txt` que tenga el total de ganancias por cada mes en el siguiente formato:

```
5-2010:933159
6-2010:1120967
7-2010:124903
```

33.4 Mezcla de números

Los archivos `a.txt` y `b.txt` tienen muchos números ordenados de menor a mayor.

Escriba un programa que cree un archivo `c.txt` que contenga todos los números presentes en `a.txt` y `b.txt` y que también esté ordenado.

No guarde los números en una estructura de datos. Vaya leyéndolos y escribiéndolos uno por uno.

Capítulo 34

Arreglos

34.1 Transmisión de datos

En varios sistemas de comunicaciones digitales los datos viajan de manera serial (es decir, uno tras otro), y en bloques de una cantidad fija de bits (valores 0 o 1). La transmisión física de los datos no conoce de esta separación por bloques, y por lo tanto es necesario que haya programas que separen y organicen los datos recibidos.

Los datos transmitidos los representaremos como arreglos cuyos valores son ceros y unos.

1. Una secuencia de bits puede interpretarse como un número decimal. Cada bit está asociado a una potencia de dos, partiendo desde el último bit. Por ejemplo, la secuencia 01001 representa al número decimal 9, ya que:

$$0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 9$$

Escriba la función `numero_decimal(datos)` que entregue la representación decimal de un arreglo de datos:

```
>>> a = array([0, 1, 0, 0, 1])
>>> numero_decimal(a)
9
```

2. Suponga que el tamaño de los bloques es de cuatro bits. Escriba la función `bloque_valido(datos)` que verifique que la corriente de datos tiene una cantidad entera de bloques:

```
>>> bloque_valido(array([0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0]))
True
>>> bloque_valido(array([0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1]))
False
```

3. Escriba la función `decodificar_bloques(datos)` que entregue un arreglo con la representación entera de cada bloque. Si un bloque está incompleto, esto debe ser indicado con el valor `-1`:

```
>>> a = array([0, 1, 0, 1])
>>> b = array([0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0])
>>> c = array([0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1])
>>> decodificar_bloques(a)
array([5])
>>> decodificar_bloques(b)
array([5, 7, 2])
>>> decodificar_bloques(c)
array([5, 7, 2, -1])
```

34.2 Series de tiempo

Una *serie de tiempo* es una secuencia de valores numéricos obtenidos al medir algún fenómeno cada cierto tiempo. Algunos ejemplos de series de tiempo son: el precio del dólar en cada segundo, el nivel medio mensual de concentración de CO_2 en el aire y las temperaturas máximas anuales de una ciudad. En un programa, los valores de una serie de tiempo se pueden guardar en un arreglo.

1. Las *medias móviles* con retardo p de una serie de tiempo son la secuencia de todos los promedios de p valores consecutivos de la serie.

   Por ejemplo, si los valores de la serie son $\{5, 2, 2, 8, -4, -1, 2\}$ entonces las medias móviles con retardo 3 son: $\frac{5+2+2}{3}$, $\frac{2+2+8}{3}$, $\frac{2+8-4}{3}$, $\frac{8-4-1}{3}$ y $\frac{-4-1+2}{3}$.

   Escriba la función `medias_moviles(serie, p)` que retorne el arreglo de las medias móviles con retardo p de la serie:

```
>>> s = array([5, 2, 2, 8, -4, -1, 2])
>>> medias_moviles(s, 3)
array([ 3,  4,  2,  1, -1])
```

2. Las *diferencias finitas* de una serie de tiempo son la secuencia de todas las diferencias entre un valor y el anterior.

   Por ejemplo, si los valores de la serie son $\{5, 2, 2, 8, -4, -1, 2\}$ entonces las diferencias finitas son: $(2-5)$, $(2-2)$, $(8-2)$, $(-4-8)$, $(-1+4)$ y $(2+1)$.

   Escriba la función `diferencias_finitas(serie)` que retorne el arreglo de las diferencias finitas de la serie:

```
>>> s = array([5, 2, 2, 8, -4, -1, 2])
>>> diferencias_finitas(s)
array([ -3,   0,   6, -12,   3,   3])
```

34.3 Creación de arreglos bidimensionales

La función `arange` retorna un arreglo con números en el rango indicado:

```
>>> from numpy import arange
>>> a = arange(12)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

A partir del arreglo `a` definido arriba, indique cómo obtener los siguientes arreglos de la manera más simple que pueda:

```
>>> # ???
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> # ???
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121]])
>>> # ???
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
>>> # ???
array([[ 0,  1,  2],
       [ 4,  5,  6],
       [ 8,  9, 10]])
>>> # ???
array([[ 11.5,  10.5,   9.5],
       [  8.5,   7.5,   6.5],
       [  5.5,   4.5,   3.5],
       [  2.5,   1.5,   0.5]])
>>> # ???
array([[100, 201, 302, 403],
       [104, 205, 306, 407],
       [108, 209, 310, 411]])
>>> # ???
array([[100, 101, 102, 103],
       [204, 205, 206, 207],
       [308, 309, 310, 311]])
```

34.4 Cuadrado mágico

Un *cuadrado mágico* es una disposición de números naturales en una tabla cuadrada, de modo que las sumas de cada columna, de cada fila y de cada

4	15	14	1
9	6	7	12
5	10	11	8
16	3	2	13

Figura 34.1: Cuadrado mágico de 4×4.

diagonal son iguales. Por ejemplo, todas las filas, columnas y diagonales del cuadrado mágico de la figura 34.1 suman 34.

Los cuadrados mágicos más populares son aquellos que tienen los números consecutivos desde el 1 hasta n^2, donde n es el número de filas y de columnas del cuadrado.

1. Escriba una función que reciba un arreglo cuadrado de enteros de $n \times n$, e indique si está conformado por los números consecutivos desde 1 hasta n^2:

```
>>> from numpy import array
>>> consecutivos(array([[3, 1, 5],
...                     [4, 7, 2],
...                     [9, 8, 6]]))
True
>>> consecutivos(array([[3, 1, 4],
...                     [4, 0, 2],
...                     [9, 9, 6]]))
False
```

2. Escriba una función que reciba un arreglo e indique si se trata o no de un cuadrado mágico:

```
>>> es_magico(array([[3, 1, 5],
...                  [4, 7, 2],
...                  [9, 8, 6]]))
False
>>> es_magico(array([[2, 7, 6],
...                  [9, 5, 1],
...                  [4, 3, 8]]))
True
```

34.5 Rotar matrices

1. Escriba la función `rotar90(a)` que retorne el arreglo `a` rotado 90 grados en el sentido contrario a las agujas del reloj:

```
>>> a = arange(12).reshape((3, 4))
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> rotar90(a)
array([[ 3,  7, 11],
       [ 2,  6, 10],
       [ 1,  5,  9],
       [ 0,  4,  8]])
```

Hay dos maneras de hacerlo: la larga (usando ciclos anidados) y la corta (usando operaciones de arreglos). Trate de hacerlo de las dos maneras.

2. Escriba las funciones `rotar180(a)` y `rotar270(a)`:

```
>>> rotar180(a)
array([[11, 10,  9,  8],
       [ 7,  6,  5,  4],
       [ 3,  2,  1,  0]])
>>> rotar270(a)
array([[ 8,  4,  0],
       [ 9,  5,  1],
       [10,  6,  2],
       [11,  7,  3]])
```

Hay tres maneras de hacerlo: la larga (usando ciclos anidados), la corta (usando operaciones de arreglos) y la astuta. Trate de hacerlo de las tres maneras.

3. Escriba el módulo `rotar.py` que contenga estas tres funciones.

```
>>> from rotar import rotar90
>>> a = array([[6, 3, 8],
...            [9, 2, 0]])
>>> rotar90(a)
array([[8, 0],
       [3, 2],
       [6, 9]])
```

34.6 Sudoku

El *sudoku* es un puzzle que consiste en llenar una grilla de 9×9 (como se ve en la figura 34.2) con los dígitos del 1 al 9, de modo que no haya ningún valor repetido en cada fila, en cada columna y en cada uno de las regiones de 3×3 marcadas por las líneas más gruesas.

El sudoku sin resolver tiene algunos de los dígitos puestos de antemano en la grilla. Cuando el puzzle ha sido resuelto, todas las casillas tienen un dígito, y entre todos satisfacen las condiciones señaladas.

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

4	2	*6*	5	*7*	1	*3*	9	*8*
8	*5*	*7*	2	*9*	3	*1*	*4*	6
1	3	*9*	*4*	6	*8*	*2*	7	*5*
9	*7*	1	*3*	*8*	*5*	6	*2*	*4*
5	4	*3*	*7*	*2*	*6*	*8*	1	9
6	*8*	2	*1*	*4*	*9*	7	*5*	*3*
7	9	*4*	*6*	3	*2*	*5*	8	*1*
2	*6*	*5*	8	*1*	4	*9*	*3*	7
3	1	*8*	9	*5*	7	*4*	6	*2*

Figura 34.2: A la izquierda: sudoku sin resolver. A la derecha: sudoku resuelto.

En un programa, un sudoku resuelto puede ser guardado en un arreglo de 9×9:

```
from numpy import array
sr = array([[4, 2, 6, 5, 7, 1, 3, 9, 8],
            [8, 5, 7, 2, 9, 3, 1, 4, 6],
            [1, 3, 9, 4, 6, 8, 2, 7, 5],
            [9, 7, 1, 3, 8, 5, 6, 2, 4],
            [5, 4, 3, 7, 2, 6, 8, 1, 9],
            [6, 8, 2, 1, 4, 9, 7, 5, 3],
            [7, 9, 4, 6, 3, 2, 5, 8, 1],
            [2, 6, 5, 8, 1, 4, 9, 3, 7],
            [3, 1, 8, 9, 5, 7, 4, 6, 2]])
```

Escriba la función `solucion_es_correcta(sudoku)` que reciba como parámetro un arreglo de 9×9 representando un sudoku resuelto, y que indique si la solución es correcta (es decir, si no hay elementos repetidos en filas, columnas y regiones):

```
>>> solucion_es_correcta(s)
True
>>> s[0, 0] = 9
>>> solucion_es_correcta(s)
False
```

34.7 Matrices especiales

1. Una matriz `a` es *simétrica* si para todo par de índices `i` y `j` se cumple que `a[i, j] == a[j, i]`.

   Escriba la función `es_simetrica(a)` que indique si la matriz `a` es simétrica o no. Cree algunas matrices simétricas y otras que no lo sean para probar su función.

2. Una matriz `a` es *antisimétrica* si para todo par de índices `i` y `j` se cumple que `a[i, j] == -a[j, i]` (note el signo menos).

   Escriba la función `es_antisimetrica(a)` que indique si la matriz `a` es antisimétrica o no. Cree algunas matrices antisimétricas y otras que no lo sean para probar su función.

3. Una matriz `a` es *diagonal* si todos sus elementos que no están en la diagonal principal tienen el valor cero. Por ejemplo, la siguiente matriz es diagonal:

$$\begin{bmatrix} 9 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

   Escriba la función `es_diagonal(a)` que indique si la matriz `a` es diagonal o no.

4. Una matriz `a` es *triangular superior* si todos sus elementos que están bajo la diagonal principal tienen el valor cero. Por ejemplo, la siguiente matriz es triangular superior:

$$\begin{bmatrix} 9 & 1 & 0 & 4 \\ 0 & 2 & 8 & -3 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

   Escriba la función `es_triangular_superior(a)` que indique si la matriz `a` es trangular superior o no.

5. No es dificil deducir qué es lo que es una matriz *triangular inferior*. Escriba la función `es_triangular_inferior(a)`. Para ahorrarse trabajo, llame a `es_triangular_superior` desde dentro de la función.

6. Una matriz es *idempotente* si el resultado del producto matricial consigo misma es la misma matriz. Por ejemplo:

$$\begin{bmatrix} 2 & -2 & -4 \\ -1 & 3 & 4 \\ 1 & -2 & -3 \end{bmatrix} \begin{bmatrix} 2 & -2 & -4 \\ -1 & 3 & 4 \\ 1 & -2 & -3 \end{bmatrix} = \begin{bmatrix} 2 & -2 & -4 \\ -1 & 3 & 4 \\ 1 & -2 & -3 \end{bmatrix}$$

   Escriba la función `es_idempotente(a)` que indique si la matriz `a` es idempotente o no.

7. Se dice que dos matrices A y B *conmutan* si los productos matriciales entre A y B y entre B y A son iguales.

   Por ejemplo, estas dos matrices sí conmutan:

$$\begin{bmatrix} 1 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} -1 & 3 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} -1 & 3 \\ 3 & 0 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 8 & 3 \\ 3 & 9 \end{bmatrix}$$

Escriba la función `conmutan` que indique si dos matrices conmutan o no. Pruebe su función con estos ejemplos:

```
>>> a = array([[ 1, 3], [3, 2]])
>>> b = array([[-1, 3], [3, 0]])
>>> conmutan(a, b)
True
>>> a = array([[3, 1, 2], [9, 2, 4]])
>>> b = array([[1, 7], [2, 9]])
>>> conmutan(a, b)
False
```

34.8 Buscaminas

El juego del buscaminas se basa en una grilla rectangular que representa un campo minado. Algunas de las casillas de la grilla tienen una mina, y otras no. El juego consiste en descubrir todas las casillas que no tienen minas.

En un programa, podemos representar un campo de buscaminas como un arreglo en el que las casillas minadas están marcadas con el valor -1, y las demás casillas con el valor 0:

```
>>> from numpy import *
>>> campo = array([[ 0,  0, -1,  0,  0,  0,  0,  0],
                   [-1,  0,  0,  0, -1,  0,  0,  0],
                   [ 0,  0,  0,  0, -1,  0,  0, -1],
                   [ 0,  0, -1,  0,  0,  0,  0,  0],
                   [ 0,  0,  0,  0,  0,  0, -1,  0],
                   [ 0, -1,  0,  0, -1,  0,  0,  0],
                   [ 0,  0, -1,  0,  0,  0,  0,  0],
                   [ 0,  0,  0,  0,  0,  0,  0,  0]])
```

1. Escriba la función `crear_campo(forma, n)`, donde `forma` es una tupla `(filas, columnas)`, que retorne un nuevo campo aleatorio con la forma indicada que tenga `n` minas.

   Hágalo en los siguientes pasos:

a) Construya un vector de tamaño `filas * columnas` que tenga `n` veces el valor -1, y a continuación sólo ceros.

b) Importe la función `shuffle` desde el módulo `numpy.random`. Esta función desordena (o «baraja») los elementos de un arreglo.

c) Desordene los elementos del vector que creó.

d) Cambie la forma del vector.

```
>>> crear_campo((4, 4), 5)
array([[-1,  0,  0,  0],
       [ 0,  0,  0,  0],
       [ 0, -1, -1,  0],
       [ 0, -1, -1,  0]])
>>> crear_campo((4, 4), 5)
array([[ 0,  0, -1,  0],
       [ 0,  0,  0, -1],
       [-1,  0,  0,  0],
       [ 0,  0, -1, -1]])
>>> crear_campo((4, 4), 5)
array([[ 0,  0,  0, -1],
       [ 0,  0, -1, -1],
       [-1,  0,  0,  0],
       [ 0,  0, -1,  0]])
```

2. Al descubrir una casilla no minada, en ella aparece un número, que indica la cantidad de minas que hay en sus ocho casillas vecinas.

   Escriba la función `descubrir(campo)` que modifique el campo poniendo en cada casilla la cantidad de minas vecinas:

```
>>> c = crear_campo((4, 4), 5)
>>> c
array([[ 0,  0, -1, -1],
       [ 0,  0, -1,  0],
       [ 0,  0,  0, -1],
       [ 0,  0,  0, -1]])
>>> descubrir(c)
>>> c
array([[ 0,  2, -1, -1],
       [ 0,  2, -1,  4],
       [ 0,  1,  3, -1],
       [ 0,  0,  2, -1]])
```