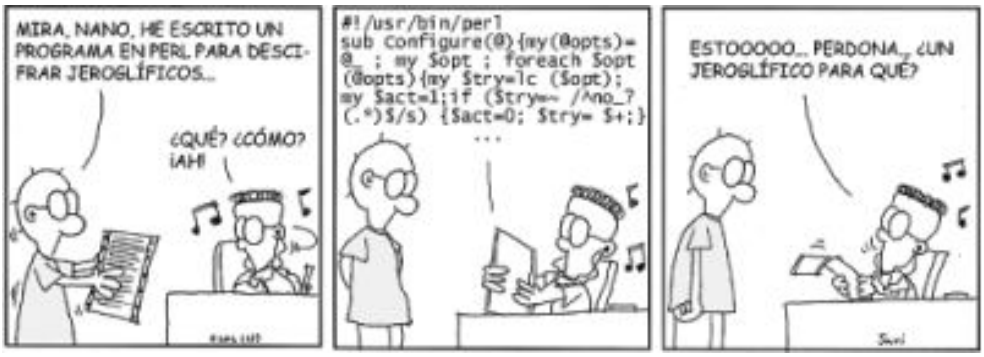
I Jornada Python en Santa Fe

Junio 2006, Argentina

python : Santa Fe 2006



Introducción

Hacia febrero del año 2006 surge en el LUGLi (Linux Users GNU/Linux, www.lugli.org.ar) el interés por organizar algunas conferencias o talleres sobre el lenguaje de programación Python. La idea fue germinando y en colaboración con la gente de PyAR, presentamos el 3 de junio a la comunidad informática de la región un evento con el objetivo de introducir este lenguaje en nuestro medio.

¿Qué es Python?

Python es un lenguaje de programación interpretado, dinámico y orientado a objetos que puede ser utilizado en diferentes desarrollos de software y en múltiples plataformas (por esto último, suele comparárselo con Java).

A pesar de lo que pueda sugerir su nombre, no es la serpiente Pitón (en inglés) quien dio su nombre a este lenguaje, sino que el mismo se debe a una aficción de su creador, Guido van Rossum, por el grupo humorista Monty Python.

La característica de ser multiplataforma es algo muy interesante para los desarrolladores. El lenguaje se encuentra soportado en Unix, GNU/Linux, Widows/DOS, Macintosh... y la lista sigue. Desde grandes mainframes a pequeñas Palm.

En la web oficial del lenguaje (www.python.org) encontrará mayor información y documentación. Así mismo, utilizando un buscador, como Google o Yahoo, encontrará mucha más información y no sólo en inglés.

La NASA, Google, Yahoo, Walt Disney y Red Hat son algunas de las grandes organizaciones que trabajan con este lenguaje. Google contrató a Guido van Rossum!!!!
Pero sin ir tan lejos, regresando a la Argentina, encontramos a Lunix, Advanta Semillas y Lambda Sistemas de quienes recientemente el diario Clarín publicó un artículo – 23/03/06 - por su producto "Fierro".

Sobre este documento

Este libro, una recopilación con motivo de la realización de la I Jornada Python en Santa Fe, recoge tres breves tutoriales sobre el lenguaje de programación Python que se pueden encontrar en la web.

Para quienes ya tienen experiencia en el lenguaje, estos documentos agregarán poco y nada a sus conocimientos, pero para quienes se inician con Python, estos recursos le serán de sumo interés y utilidad.

Esperamos lo aprendan y disfruten.

Indice

Breve introducción al lenguaje Python

Este documento pretende ser una pequeña introducción al lenguaje Python. Sólo se explican, muy brevemente, los rasgos generales de Python. Para una mayor explicación de los pormenores del lenguaje se recomienda ***Python Instantaneo***.

por Arturo Suelves Albert (arturosa@lycos.es)

Sentencias y comentarios:

- En Python las sentencias se agrupan mediante el indentado (ni con llaves como en C ni con BEGIN/END como en pascal), las lineas que tienen la misma "cantidad" de indentado son del mismo bloque.
- Cada linea del programa es una sentencia separada
- Los comentarios comienzan con **#** , llegan sólo hasta el final de la linea y son ignorados en la ejecución.

Ejemplo:

```
# Esta linea es un comentario
Aqui comineza el 1 bloque de sentencias:
    primera sentencia
    segunda sentencia
       Aqui comineza el 2 bloque de sentencias:
           primera sentencia
           segunda sentencia #Esto es un comentario hasta el fin de linea
           tercera sentencia
    Esta sentencia no pertenece al 2 bloque, sino al 1 bloque
Sentencia
Sentencia
# Esta linea es un comentario
Aqui comienza otro bloque de sentencias:
    priemra sentencia
    segunda sentencia
```

Tipos y variables

- Los tipos básicos disponibles son: *enteros*, *floats* (reales), *strings* (cadenas), *lists* (listas).
- En python los `strings` pueden ir delimitados por comillas simples o dobles.
- Las variables se declaran y asignan cuando se usan (y sin declarar el tipo).
- Cuidado con usar una variable antes de asignarle un valor.
- El operador de asignación es = (cuidado, el operador booleano de comparación no es **=** sino **==**).
- Python permite asignaciones de variables multiples.

Ejemplos:

- Algunos enteros: 1
  23
  456
- Algunos reales 1.0
  24.34
- String o cadenas “Hola mundo”
  'Hola mundo'
- List o listas [1 , 2 , 3]
  [“Hola” , “mundo” , 123]

En python las listas van encerradas entre `[ ]` y los valores separados por `,`

```
variable1 = 1
variable2 = 2
variable1,variable2 = 1,2 #Asignación de variables múltiple
variable_autor = "arturo suelves" #Asignación de un string
variable_lista = [1, 2, 3] #Asignación de una lista
```

Expresiones

- Cualquier expresión con enteros puede ser construida con los operadores estandar (+, - ,* , /) y con la agrupación mediante paréntesis.
- Se admite la concatenación de cadenas y listas con el operador +
- Se pueden construir substrings de un string mediante índices.(cuidado: se puede acceder al valor de un elemento del string pero no cambiarlo)
- Se puede acceder a los valores de una lista mediante índices (a diferencia de con los strings los elementos de una lista si se pueden cambiar)

Ejemplos:

```
1+2*3 # El resultado es 7

(1+2)*3 # El resultado es 9

var_autor = "arturo suelves" # Vamos a extraer substrings de este string

var_iniciales=var_autor[0]+var_autor[7] # El resultado es as

var_nombre = var_autor[0:6] #El resultado es arturo

var_lista=["Hola","a","todo","el","mundo"]

elemento_de_lista=var_lista[2] #El resultado es todo

elementos_de_lista=var_lista[0:2] #El resultado es ["Hola","a"]

var_lista[2] = "casi todo" # El resultado es var_lista=
                                    ["Hola","a","casi todo","el","mundo"]
```

Expresiones y operadores booleanos

- Cualquier valor distinto a `""`,`[]` y `0` representa VERDADERO (`true`), sino representa FALSO (`false`).
- Las expresiones booleanas se pueden construir con los operadores `and`, `or` ,`not` y con sus agrupaciones mediante `( )`
- El operador booleano de comparación es **==** (cuidado, el operador de asignación es =)

Ejemplos:

```
1 > 2 #Falso 0

2>1 # Verdadero 1

"A" >= "B" and "C" <= "D" # Falso 0

not ("A" >= "B" and "C" <= "D") # Verdadero 1
```

Control de flujo(if , for)

Sentencia IF:

```
if valor < 0:
    print 'Valor es menor que cero'
else:
    print 'Valor es mayor o igual que cero'
```

Se pueden concatenar varios IF con la estructura IF..ELIF...ELIF..

```
if valor < 0:
    print 'Valor es menor que cero'
elif valor == 0:
    print 'Valor es cero'
elif valor == 1:
    print 'Valor es uno'
else:
    print 'Valor es mayor que uno'
```

Sentencia FOR:

```
for valor in [valor1,valor2,valor3]:
    print valor
```

En Python los bucles for siempre cogen su rango de los valores de una lista. Para poder hacer bucles sobre valores numéricos tenemos la función `range(n)`, siendo `n` el número de valores.

Ejemplo:

```
for valor in range(10):

    print valor
```

Esto imprime 10 valores, desde el 0 hasta el 9.

Funciones

- Las funciones se declaran con la palabra reservada def.
- Siempre se pasan los parámetros por referencia
- El valor retornado por la función lo devuelve la sentencia return
- Se puede asignar a una variable el valor retornado por una función

Ejemplos:

```
def producto(x,y):
  resultado=x*y
  return resultado

def cuadrado(x):
  return product(x,x)

contador= cuadrado(2) #el valor de contador es 4
```

Programas

Las funciones más útiles se ponen en módulos,que son en realidad ficheros de texto con código Python. Estos pueden ser importados y utilizados en tus programas.

Python incorpara muchos módulos para ayudar a crear programas.Consulta en la ayuda de Python el apartado de módulos.

Por ejemplo, para convertir cadenas de minúsculas a mayúsculas podemos usar del modulo `string` la función `upper`:

```
import string # importamos el modulo string

def convertir_mM(cadena):
      #  devuelve la cadena convertida en mayusculas
      return string.upper(cadena)

if __name__ == "__main__": print convertir_mM('aEIou')
```

Esta última línea permite indicar al interprete de python que este fichero sea ejecutable (es decir, que no esta siendo importado por ningún otro módulo o programa). Después de los 2 puntos puedes poner el código que quieras que se ejecute al lanzar tu programa.

Para ejecutar tu programa sólo debes de guardar tu código en un fichero e invocar al intérprete de python pasando el nombre del fichero (y su trayectoria) como argumento.

Ejemplo:

```
python mi_fichero
```

Python instantáneo

Esto es un curso intensivo de introducción mínima al lenguaje de programación Python . Para obtener más información, echa un vistazo a la documentación de la web de Python, `http://www.python.org`, especialmente el tutorial. Si te preguntas por qué debería interesarte, mira la página `http://wiki.python.org/moin/LanguageComparisons`, en la que aparece Python comparado con otros lenguajes.

1. Lo básico

Para empezar, piensa en Python como pseudo-código. Esto es casi cierto. Las variables no tienen tipo, así que no hay que declararlas. Aparecen cuando se les asigna algo y desaparecen al dejar de usarlas. La asignación se realiza mediante el operador `=`. Para comprobar la igualdad se utiliza el operador `==` . Se puede asignar varias variables a la vez:

```
x,y,z = 1,2,3
primero, segundo = segundo, primero
a = b = 123
```

Para definir bloques de código, se utiliza el sangrado (o indentación) solamente (nada de BEGIN/END ni llaves). Éstas son algunas estructuras de control comunes:

```
if x < 5 or (x > 10 and x < 20):
    print "El valor es correcto."

if x < 5 or 10 < x < 20:
    print "El valor es correcto."

for i in [1,2,3,4,5]:
    print "Pasada n° ", i

x = 10
while x >= 0:
    print "x todavía no es negativo."
    x = x-1
```

Los dos primeros ejemplos son equivalentes.

La variable de índice en el bucle for recorre los elementos de una lista (escrita como en el ejemplo). Para realizar un bucle for "normal" (es decir, contando), utilice la función de serie range().

```
# Mostrar los valores de 0 a 99 inclusive.
for valor in range(100):
    print valor
```

(La línea que comienza por "# " es un comentario y el intérprete le hace caso omiso)

Bien, ahora ya sabes suficiente para implementar cualquier algoritmo en Python. Vamos a añadir algo de interacción básica . Para obtener entrada del usuario, de un indicador de texto, utiliza la función de serie input.

```
x = input("Introduzca un número:")
print "El cuadrado de ese número es:", x*x
```

La función `input` muestra la solicitud dada (que podría estar vacía) y permite que el usuario introduzca cualquier valor Python válido. En este caso esperábamos un número, si se introduce algo diferente (una cadena, por ejemplo), el programa falla. Para evitarlo necesitaríamos algo de comprobación de errores. No voy a entrar en ese tema ahora, valga decir que si quiere guardar lo que el usuario ha introducido textualmente como un cadena (para que se pueda introducir cualquier cosa), utilice la función `raw_input`. Si desea convertir la cadena de entrada *s* a un *entero*, podría utilizar `int(s)`.

Nota: Si desea introducir una cadena con `input` , el usuario debe escribir las comillas explícitamente. En Python, las cadena pueden encerrarse entre comillas simples o dobles.

Así que tenemos cubiertas las estructuras de control, la entrada y la salida. Ahora necesitamos estructuras de datos. Las más importantes son las listas y los diccionarios . Las listas se escriben entre corchetes, y se pueden (por supuesto) anidar:

```
nombre = ["Cleese", "John"]
x = [[1,2,3],[y,z],[[[]]]]
```

Una de las ventajas de las listas es que se puede acceder a sus elementos por separado o en grupos, mediante indexado y corte . El indexado se realiza (como en muchos otros lenguajes) añadiendo el índice entre corchetes a la lista (observa que el primer elemento es el 0).

```
print nombre[1], nombre[0]
Muestra "John Cleese"
nombre[0] = "Palin"
```

El corte es casi como el indexado, pero se indican los índices de inicio y fin del resultado, con dos puntos (":") de separación:

```
x = ["magro","magro","magro","magro","magro","huevos","and","magro"]

print x[5:7]
Muestra la lista ["huevos","and"]
```

Observa que el índice final no se incluye en el resultado. Si se omite uno de los índices se supone que se quiere obtener todo en la dirección correspondiente. Esto es, `lista[:3]` quiere decir "cada elemento desde el principio de lista hasta el elemento 3, no incluido". (se podría decir en realidad el elemento 4, ya que contamos desde 0... bueno). `lista[3:]` significaría, por otra parte "cada elemento de lista , empezando por el 3 (inclusive), hasta el último inclusive". Se pueden utilizar números negativos para obtener resultados muy interesantes: `lista[-3]` es el tercer elemento desde el final de la lista...

Ya que estamos en el tema del indexado, puedes encontrar interesante que la función de serie `len` dé la longitud de una lista.

Y ahora, ¿qué pasa con los diccionarios? Para ser breves, son como listas, pero su contenido no está ordenado. Y ¿cómo se indexan, entonces? Bueno, cada elemento tiene una clave o "nombre" que se utiliza para buscar el elemento, como en un diccionario de verdad. Un par de diccionarios de ejemplo:

```
{ "Alicia" : 23452532, "Boris" : 252336, "Clara" : 2352525 }

persona = { 'nombre': "Robin", 'apellido': "Hood",
            'trabajo u ocupación': "Ladrón" }
```

Ahora, para obtener la ocupación de persona, utilizamos la expresión `persona["trabajo u ocupación"]` . Si le queremos cambiar el apellido, escribiremos:

```
persona['apellido'] = "de Locksley"
```

Simple, ¿no? Como las listas, los diccionarios pueden contener otros diccionarios. O listas, ya que nos ponemos. Y naturalmente, también las listas pueden contener diccionarios. De este modo, se pueden conseguir estructuras de datos bastante avanzadas.

2. Funciones

Próximo paso: Abstracción. Queremos dar un nombre a un trozo de código y llamarlo con un par de parámetros. En otras palabras, queremos definir una función (o "procedimiento"). Es fácil. Utilice la palabra clave `def` así:

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

print cuadrado(2)
Muestra 4
```

Para los que lo entiendan: *Todos los parámetros en Python se pasan por referencia* (como, por ejemplo, en Java). Para los que no, no se preocupen.

Python tiene todo tipo de lujos, como argumentos con nombre y argumentos por omisión y puede manejar un número variable de argumentos para una función. Para obtener más información, consulta la sección 4.7 del tutorial de Python.

Si sabe utilizar las funciones en general, esto es lo que necesita saber sobre ellas en Python, básicamente (ah, sí, la palabra clave `return` detiene la ejecución de la función y devuelve el resultado indicado).

Algo que podría resultar interesante conocer, sin embargo, es que las funciones son valores en Python. Así que, si tiene una función como cuadrado, podría hacer cosas como:

```
cosa = cuadrado
cosa(2)
Muestra 4
```

Para llamar a una función sin argumentos debes recordar escribir `hazlo()` y no `hazlo`. La segunda forma sólo devuelve la función en sí, como valor (esto vale también para los métodos de los objetos... lee lo siguiente).

3. Objetos y cosas...

Supongo que sabes cómo funciona la programación orientada a objetos (de otro modo, esta sección podría resultar un poco confusa, pero no importa, empieza a jugar con los objetos :)). En Python se definen las clases con la palabra clave (¡sorpresa!) `class`, de este modo:

```
class Cesta:
    # Recuerde siempre el argumento self
    def __init__(self,contenido=None):
        self.contenido = contenido or []
    def añadir(self,elemento):
        self.contenido.append(elemento)
    def muestra_me(self):
        resultado = ""
        for elemento in self.contenido:
            resultado = resultado + " " + `elemento`
        print "Contiene:"+resultado
```

Cosas nuevas:

1. Todos los métodos (funciones de un objeto) reciben un argumento adicional al principio de la lista de argumentos, que contiene el propio objeto. Este argumento, por convención, se suele llamar self (que significa 'uno mismo'), como en el ejemplo.
2. A los métodos se los llama de este modo: `objeto.método(arg1,arg2)`.
3. Algunos nombres de métodos, como `__init__` están predefinidos, y significan cosas especiales. `__init__` es el nombre del constructor de la clase, es decir, es la función a la que se llama cuando creas una instancia.
4. Algunos argumentos son opcionales y reciben un valor dado (según lo mencionado antes, en la sección de funciones). Esto se realiza escribiendo la definición así:

   ```
   def magro(edad=32): ...
   ```

   Aquí, se puede llamar a magro con uno o cero parámetros. Si no se pone ninguno, el parámetro edad tendrá el valor 32.
5. "Lógica de cortocircuito." Esto es un punto... Ver más tarde.
6. Las comillas invertidas convierten un objeto en su representación como cadena (así que si elemento contiene el número 1, `elemento` es lo mismo que "1" mientras que 'elemento' es una cadena literal).
7. El signo más `+` se utiliza también para concatenar listas. Las cadenas son sólo listas de caracteres (lo que significa que se puede utilizar indexado, corte y la función `len` en ellas).

Ningún método o variable miembro es protegido (ni privado) en Python. *La encapsulación es, en su mayoría, cuestión de estilo al programar*.

Retomando el tema de la lógica de cortocircuito...

Todos los valores de Python se pueden utilizar como valores lógicos. Algunos, los más "vacíos", como `[]`, `0`, `""` y `None` representan el valor lógico "`falso`", mientras el resto (como `[0]`, `1` or `"Hola, mundo"`) representan el valor lógico "`verdadero`".

Las expresiones lógicas como `a and b` se evalúan de este modo: Primero, se comprueba si `a` es verdadero. Si no, simplemente se devuelve su valor. Si sí, simplemente se devuelve `b` (que representa el valor lógico de la expresión). La lógica correspondiente a a or b es: Si a es verdadero, devolver su valor. Si no, devolver b.

Este mecanismo hace que `and` y `or` se comporten como los operadores lógicos que supuestamente implementan, pero también permite escribir expresiones condicionales muy curiosas. Por ejemplo, el código

```
if a:
    print a
else:
    print b
```

Se puede sustituir por:

```
print a or b
```

De hecho, esto es casi un 'deje' en Python, así que mejor irse acostumbrando. Esto es lo que hacemos en el método `Cesta.__init__` . El argumento contenido tiene el valor por defecto `None` (que es, entre otras cosas, falso). Por lo tanto, para comprobar si tenía valor, podríamos escribir:

```
if contenido:
    self.contenido = contenido
else:
    self.contenido = []
```

Por supuesto, ahora conocemos un método mejor, Y, ¿por qué no le damos el valor por omisión `[]` para empezar? Por el modo en que funciona Python, esto daría a todas las `Cestas` la misma lista vacía como contenido por omisión. Tan pronto como se empezara a llenar una de ellas, todas tendrían los mismos elementos y el valor por omisión dejaría de ser vacío... Para informarse sobre el tema, lee la documentación y busca la diferencia entre identidad e igualdad.

Otro modo de hacer lo anterior es:

```
def __init__(self, contenido=[]):
    self.contenido = contenido[:]
```

¿Adivinas cómo funciona esto? En lugar de utilizar la misma lista vacía siempre, utilizamos la expresión `contenido[:]` para hacer una copia (hacemos un corte que contiene toda la lista).

Así que, para hacer realmente una Cesta y utilizarla (es decir, llamar a alguno de sus métodos) haríamos algo así:

```
b = Cesta(['manzana','naranja'])
b.añadir("limón")
b.muestra_me()
```

Hay más métodos mágicos además de `__init__`. Uno de ellos es `__str__`, que define el aspecto que quiere tener el objeto si se le trata como una cadena. Lo utilizaríamos en nuestra cesta en lugar de `presenta_me` :

```
def __str__(self):
    resultado = ""
    for elemento in self.contenido:
        resultado = resultado + " " + `elemento`
    return "Contiene:"+resultado
```

Y, si quisiéramos mostrar la cesta `b`, simplemente diríamos: `print b`

La herencia se realiza de este modo:

```
class CestaMagro(Cesta):
    # ...
```

Python permite la herencia múltiple, así que puede indicar varias super-clases entre los paréntesis, separadas por comas. Las clases se instancian así: `x = Cesta()`. Los constructores se definen, como dije, implementando la función miembro especial `__init__`. Pongamos que `CestaMagro` tuviera un constructor `__init__(self,tipo)`. Podría realizar una cesta de magro así: `y = CestaMagro("manzanas")` .

Si necesitase llamar al constructor de una super-clase desde el constructor H de `CestaMagro`, lo haría así: `Cesta.__init__(self)` . Observe que, además de proporcionar los parámetros normales, debe proporcionar explícitamente `self`, ya que `__init__` de la super-clase no sabe con qué instancia está tratando.

Para obtener más información sobre las maravillas de la programación orientada a objetos en Python, mire la sección 9 del tutorial.

4. Truco mental Jedi

(Esta sección no es necesario en absoluto leerla para empezar a aprender Python)

¿Te gustan los ejercicios mentales? Si es así, si eres realmente osado, deberías echarle un vistazo al ensayo de Guido van Rossum sobre metaclases . Si, por el contrario, prefieres que el cerebro no te explote, igual te satisface este truquito.

Python utiliza espacios de nombres dinámicos (no léxicos). Esto quiere decir que si tienes una función como ésta:

```
def zumo_naranja():
    return x*2
```

... donde una variable (en este caso x) no está ligada a un argumento, y no se le asigna un valor desde dentro de la función, Python utilizará el valor que tenga cuando se llama a la función. En este caso:

```
x = 3
zumo_naranja()
Devuelve 6

x=1
zumo_naranja()
Devuelve 2
```

Normalmente, éste es el comportamiento deseado (aunque el ejemplo es un poco rebuscado, pues es raro acceder a las variables de este modo). Sin embargo, a veces puede ser útil tener un espacio de nombres estático, es decir, guardar algún valor del entorno en que se crea la función. El modo de hacer esto en Python es por medio de los argumentos por omisión.

```
x = 4
def zumo_manzana(x=x):
    return x*2
```

Aquí, al argumento `x` se le asigna un valor por defecto que coincide con el valor de la variable `x` en el instante en que la función es definida. Por lo tanto, siempre que nadie proporcione un argumento para la función, funcionará así:

```
x = 3
zumo_manzana():
Devuelve 8
x = 1
zumo_manzana():
Devuelve 8
```

Concluyendo: El valor de `x` no cambia. Si esto fuese todo lo que queríamos, podríamos limitarnos a escribir

```
def zumo_tomate():
    x = 4
    return x*2
```

incluso

```
def zumo_zanahoria():
    return 8
```

Sin embargo, lo importante es que el valor de `x` se toma del entorno en el instante en que se define la función. ¿Qué utilidad tiene esto? Tomemos un ejemplo: Una función compuesta.

Queremos una función que funcione así:

```
from math import sin, cos
sincos = componer(sin,cos)
x = sincos(3)
```

Donde componer es la función que queremos realizar y `x` tiene el valor `-0.836021861538`, que es lo mismo que `sin(cos(3))`. Y ¿cómo lo hacemos?

Observa que estamos utilizando funciones como argumentos y eso ya es un truco en sí mismo. Obviamente, `componer` toma dos funciones como parámetros y devuelve una función que a su vez toma un parámetro. Un esqueleto de la solución podría ser:

```
def componer(fun1, fun2):
    def interior(x):
        # ...
    return interior
```

Nos tentaría poner `return fun1(fun2(x))` dentro de la función interior y dejarlo tal cual. No, no y no. Eso tendría resultados muy extraños. Imagina la siguiente situación:

```
from math import sin, cos
def fun1(x):
    return x + " mundo"
def fun2(x):
    return "Hola,"
sincos = componer(sin,cos)  # Versión incorrecta
x = sincos(3)
```

Y bien, ¿qué valor tendría `x`? Correcto: "Hola, mundo". Y ¿por qué? Porque cuando se la llama, toma los valores de `fun1` y `fun2` del entorno, no los que andaban por ahí cuando se creó. Para conseguir una función correcta, sólo hay que utilizar la técnica descrita anteriormente:

```
def componer(fun1, fun2):
    def interior(x, fun1=fun1, fun2=fun2):
        return fun1(fun2(x))
    return interior
```

Ahora sólo nos queda esperar que nadie proporcione a la función resultante más de un argumento, ya que eso nos rompería los esquemas. Y, a propósito, como no necesitamos el nombre interior y sólo contiene una expresión, podemos utilizar una función anónima, utilizando la palabra clave lambda:

```
def componer(f1, f2):
    return lambda x, f1=f1, f2=f2: f1(f2(x))
```

Espartano, pero claro...y si no ha entendido nada, no se preocupe. Al menos le ha convencido de que Python es más que "un lenguajillo para scripts"...

5. Y ahora...

Sólo unas cosillas para terminar. Las funciones y clases más útiles se ponen en módulos, que son en realidad ficheros de texto legible con código Python. Puede importarlos y utilizarlos en sus propios programas. Por ejemplo, para utilizar el método `split` (trocear) del módulo estándar `string` (cadena), puede hacer estas dos cosas:

```
import string
x = string.split(y)
```

o...

```
from string import split
x = split(y)
```

Para obtener más información sobre la biblioteca de módulos estándar, echa un vistazo a `www.python.org/doc/lib`. Contiene un montón de cosas útiles.

Todo el código del módulo/script se ejecuta cuando se importa. Si quiere que su programa sea tanto un módulo importable como un script ejecutable, puede añadir algo así al final:

```
if __nombre__ == "__main__": ejecutar()
```

Es un modo mágico de decir que si el módulo se ejecuta como un script ejecutable (es decir, que no esta siendo importado por otro script o módulo), se debe ejecutar la función ejecutar. Por supuesto, puede hacer cualquier cosa tras los dos puntos... :)

Y, si desea hacer un script ejecutable en UNIX, escriba esto como primera línea para hacer que el script se ejecute sin llamar a python explícitamente:

```
#!/usr/bin/env python
```

Finalmente, una breve mención a un concepto importante: Las **excepciones**. Algunas operaciones (como dividir por cero o leer de un archivo inexistente) causan una condición de error o excepción. Puede incluso generar las suyas propias y lanzarlas en los momentos adecuados.

Si no se hace nada con la excepción, el programa termina y muestra un mensaje de error. Esto se puede evitar con una construcción `try/except`. Por ejemplo:

```
def dividirSeguro(a,b):
    try:
        return a/b
  except ZeroDivisionError:
        return None
```

`ZeroDivisionError` es una excepción estándar. En este caso, se podría haber mirado si `b` era cero, pero hay muchos casos en los que no es posible. Además, si no tuviéramos la cláusula `try in dividirSeguro`, haciéndola de este modo una función arriesgada, podríamos hacer algo como:

```
try:
    dividirInseguro(a,b)
except ZeroDivisionError:
    print "Se ha intentado dividir por cero en dividirInseguro"
```

En casos en los que normalmente no debería haber problemas concretos, pero podrían ocurrir, la utilización de excepciones permite evitar tediosas comprobaciones.

Bueno, eso es todo. Espero que aprendieras algo. Ahora, a jugar. Y recuerde el lema de Python para aprender: "Que las fuentes te acompañen" (léase: Leer todo el código al que se le pueda echar las manos encima). Para arrancar, aquí hay un ejemplo. Es el conocido algoritmo de Hoare, `QuickSort`.

Merece la pena resaltar una cosa sobre el ejemplo. La variable `done` controla si la `partition` ha terminado o no de recorrer los elementos. Así que cuando uno de los dos bucles internos desea terminar la secuencia de intercambio completa, pone `done` a 1 y sale él mismo mediante `break`. ¿Por qué utilizan `done` los bucles internos? Porque, cuando el primer bucle interno finaliza con un `break` , que el siguiente bucle deba rearrancar depende de si el bucle principal ha finalizado, esto es, si `done` se ha puesto a 1 :

```
while not done:
    while not done:
        # Se repite hasta un 'break'
    while not done:
        # Sólo se ejecuta si el primero no puso "done" a 1
```

Una versión equivalente, posiblemente más clara, pero en mi opinión menos elegante, sería:

```
while not done:
    while 1:
        # Se repite hasta un 'break'
    if not done:
        while 1:
            # Sólo se ejecuta si el primero no puso "done" a 1
```

La única razón por la que he utilizado la variable `done` en el primer bucle ha sido que me gustaba conservar la simetría entre los dos. De este modo, se podría invertir el orden y el algoritmo todavía funcionaría.

Se pueden encontrar más ejemplos en la página `tidbit` de Joe Strout.
(`http://www.strout.net/python/tidbits.html`).

Python para No-Programadores

http://honors.montana.edu/~jjc/easytut/easytut_es/

1. Introducción

El Tutor de Python para No-programadores es un tutor diseñado como una introducción al lenguaje de programación Python. Esta guía es para alguien que no tiene experiencia previa en programación.

Si ha programado en otros lenguajes, le recomiendo usar The Python Tutorial escrito por Guido van Rossum.

Este documento (en inglés) está disponible en LATEX, HTML, PDF, and Postscript. Vaya a `http://www.honors.montana.edu/~jjc/easytut/` para ver todos estos formatos.

Si tiene preguntas o comentarios (en inglés) envíeme un mensaje a `jjc@iname.com` todas las preguntas y comentarios acerca de este manual son bienvenidos. Trataré de contestar todas las preguntas lo mejor que pueda.

Gracias a James A. Brown por escribir la mayor parte de la información de instalación para Windows. Gracias también a Elizabeth Cogliati por quejarse lo suficiente :) acerca del tutor original, (que es imposible de usar por un no-programador) por leer las pruebas y por sus muchas ideas y comentarios. Gracias también a Joe Oppegaard por escribir todos los ejercicios. Gracias a todos aquellos que olvidé mencionar.

1.1 Primero lo Primero

Así que nunca antes ha programado. A medida que avancemos en este tutor intentaré enseñarle cómo programar. Realmente sólo hay un modo de aprender a programar. Usted debe leer código fuente y escribir código fuente. Le mostraré mucho código. Usted debería escribir en el interpretador el código que le muestro para que vea qué sucede. Juegue y haga cambios. Lo peor que puede pasar es que no funcione. El código fuente estará en el siguiente tipo de letra:

```
##Python es facil de aprender
print "Hola, Mundo!"
```

Así que será muy fácil de distinguir de lo demás del texto. Para confundir las cosas también incluiré en el mismo tipo de letra la respuesta de la computadora.

Ahora pasemos a cosas más importantes. Para programar en Python necesita el software Python. Si no lo tiene todavía vaya a `http://www.python.org/download/` y obtenga la versión adecuada para su computadora y sistema operativo. Descárguelo, lea las instrucciones e instálelo.

1.2 Instalando Python

Primero necesita descargar el archivo adecuado para su computadora de `http://www.python.org/download`. Vaya al eslabón 2.0 (o más reciente) y obtenga el instalador para windows, si usa Windows o el rpm o el código fuente si utiliza Unix.

El instalador de Windows descargará el archivo. Puede correr el archivo con un doble click sobre el ícono que fue descargado. Entonces procederá la instalación.

Si obtiene el código fuente para Unix, asegúrese de compilarlo con la extensión tk si desea usar IDLE.

1.3 Modo Interactivo

Vaya a IDLE (tambié llamado la interfase gráfica de Python). Deberá ver una ventana con un texto similar al siguiente:

```
Python 2.0 (#4, Dec 12 2000, 19:19:57)
[GCC 2.95.2 20000220 (Debian GNU/Linux)] on linux2
Type "copyright", "credits" or "license" for more information.
IDLE 0.6 -- press F1 for help
>>>
```

El símbolo >>> es el modo de Python de decir que está en modo interactivo. En modo interactivo lo que escriba será interpretado inmediatamente. Intente escribir 1+1. Python responderá con un 2. El modo interactivo le permite experimentar y ver lo que Python hará. Si alguna vez necesita jugar con enunciados nuevos de Python entre al modo interactivo y experimente con ellos.

1.4 Creando y Corriendo Programas

Arranque IDLE si no lo ha hecho ya. Abra el menú File y luego New Window. En esta ventana escriba lo siguiente:

```
print "Hola, Mundo!"
```

Primero guarde el programa. Vaya a File y luego Save. Guárdelo com hola.py. (Si lo desea, puede guardarlo en algún otro directorio que no sea el definido por IDLE.) Ahora que ya está guardado, puede correr.

Ahora corra el programa yendo a `Edit` y luego `Run script`. Esto generará `Hola, Mundo!` en la ventana llamada `*Python Shell*` .

¿Todavía confundido? Intente con este tutor para IDLE (en inglés) en `http://hkn.eecs.berkeley.edu/~dyoo/python/idle_intro/index.html`

1.5 Usando Python desde la línea de comandos

(Si no desea usar Python desde la línea de comandos, no tiene obligación de hacerlo, sólo use IDLE.) Para entrar a modo interactivo teclee python sin argumentos en la línea de comandos. Para correr un programa, cree el programa con un editor de textos (Emacs tiene un buen modo de python) y luego córralo con `python nombredelprograma`.

2. Hola, Mundo

2.1 Lo que debe saber

Debe saber cómo escribir programas en un editor de textos o IDLE, guardarlos a disco (duro o flexible) y cómo correrlos una vez que están guardados.

2.2 Imprimiendo

Desde el principio del tiempo, los tutores de programación han comenzado con un pequeño programa llamado Hola, Mundo! Así que aquí está:

```
print "Hola, Mundo!"
```

Si está usando la línea de comandos para correr programas, escríbalo en un editor de textos, guárdelo como `hola.py` y córralo con

```
python hola.py
```

De otro modo, entre a IDLE, abra una nueva ventana y cree el programa como en la sección 1.4.

Cuando este programa corre, esto es lo que imprime en la pantalla:

```
Hola, Mundo!
```

Ahora, no repetiré esto cada vez, pero cuando muestro un programa, recomiendo que lo escriba y lo corra. Yo aprendo mejor cuando escribo y probablemente usted también.

Ahora este es un programa más complicado:

```
print "Jack y Jill subieron a la colina"
print "a traer una cubeta de agua;"
print "Jack se cayo y se rompio una pierna,"
print "y Jill vino rodando tras el."
```

Cuando usted corre este programa, éste imprime:

```
Jack y Jill subieron a la colina
a traer una cubeta de agua;
Jack se cayo y se rompio una pierna,
y Jill vino rodando tras el.
```

Cuando la computadora corre este programa primero ve la línea:

```
print "Jack y Jill subieron a la colina"
```

así que la computadora imprime: `Jack y Jill subieron a la colina`

Luego la computadora baja a la siguiente línea y ve:

```
print "a traer una cubeta de agua;"
```

así que la computadora imprime en la pantalla: `a traer una cubeta de agua;`

La computadora continúa viendo cada renglón, ejecuta el comando y luego va al siguiente renglón. La computadora continúa ejecutando comandos hasta que llega al final del programa.

2.3 Expresiones

Aquí está otro programa:

```
print "2 + 2 es", 2+2
print "3 * 4 es", 3 * 4
print 100 - 1, " = 100 - 1"
print "(33 + 2) / 5 + 11.5 = ",(33 + 2) / 5 + 11.5
```

Y aquí está la respuesta del programa cuando corre:

```
2 + 2 es 4
3 * 4 es 12
99  = 100 - 1
(33 + 2) / 5 + 11.5 =  18.5
```

Como puede ver, Python puede convertir su computadora de miles de dólares en una calculadora de bolsillo de 5 dólares.

Python tiene seis operaciones básicas:

Operación	*Símbolo*	*Ejemplo*
Exponenciación	**	5 ** 2 == 25
Multiplicación	*	2 * 3 == 6
División	/	14 / 3 == 2
Residuo (resto)	%	14 % 3 == 2
Adición	+	1 + 2 == 3
Sustracción	-	4 – 3 == 1

Note que la división sigue la regla, si no hay decimales al comienzo, tampoco habra decimales al final. (Nota: Esto tal vez cambie en 2.3) El siguiente programa muestra esto:

```
print "14 / 3 = ",14 / 3
print "14 % 3 = ",14 % 3
print
print "14.0 / 3.0 =",14.0 / 3.0
print "14.0 % 3.0 =",14 % 3.0
print
print "14.0 / 3 =",14.0 / 3
print "14.0 % 3 =",14.0 % 3
print
print "14 / 3.0 =",14 / 3.0
print "14 % 3.0 =",14 % 3.0
print
```

Con la salida:

```
14 / 3 =  4
14 % 3 =  2

14.0 / 3.0 = 4.66666666667
14.0 % 3.0 = 2.0

14.0 / 3 = 4.66666666667
14.0 % 3 = 2.0

14 / 3.0 = 4.66666666667
14 % 3.0 = 2.0
```

Note cómo Python da diferentes respuestas para algunos problemas dependiendo del uso de punto decimal.

El orden de las operaciones es el mismo que en matemáticas:

1. paréntesis `()`
2. exponentes `**`
3. multiplicación `*`, división `\`, y residuo `%`
4. adición `+` y sustracción `-`

2.4 Hablando con humanos (y otros seres inteligentes)

Con frecuencia, usted hace algo complicado y puede ser que luego no recuerde lo que hizo. Cuando esto sucede, probablemente haya que comentar el programa. Un comentario es una nota para usted y otros programadores explicando lo que sucede. Por ejemplo:

```
#No es exactamente PI, pero lo simula muy bien
print 22.0/7.0
```

Note que el comentario comienza con un `#`. Los comentarios aclaran los puntos complicados a cualquiera que lea el programa, uno mismo inclusive.

2.5 Ejemplos

Eventualmente, cada capítulo contendrá ejemplos de los aspectos de programación abordados en el capítulo. Por lo menos, debería leerlos para ver si los entiende. Si no los entiende, tal vez quiera escribirlos para ver qué pasa. Juegue con los ejemplos, cámbielos y vea lo que pasa.

Dinamarca.py

```
print "Algo esta podrido en el estado de Dinamarca."
print "                -- Shakespeare"
```

Salida:

```
Algo esta podrido en el estado de Dinamarca.
                -- Shakespeare
```

Escuela.py

```
#Esto no es completamente cierto fuera de los EE UU
# y esta basado en mis borrosos recuerdos de cuando era joven
print "Primer Grado"
print "1+1 =",1+1
print "2+4 =",2+4
print "5-2 =",5-2
print
print "Tercer Grado"
print "243-23 =",243-23
print "12*4 =",12*4
print "12/3 =",12/3
print "13/3 =",13/3," R ",13%3
print
print "Secundaria"
print "123.56-62.12 =",123.56-62.12
print "(4+3)*2 =",(4+3)*2
print "4+3*2 =",4+3*2
print "3**2 =",3**2
print
```

Impresión en pantalla:

```
Primer Grado
1+1 = 2
2+4 = 6
5-2 = 3

Tercer Grado
243-23 = 220
12*4 = 48
12/3 = 4
13/3 = 4  R  1

Secundaria
123.56-62.12 = 61.44
(4+3)*2 = 14
4+3*2 = 10
3**2 = 9
```

2.6 Ejercicios

Escriba un programa que imprima su nombre completo y su cumpleaños como hileras separadas.

Escriba un programa que use las 6 operaciones básicas.

3. ¿Quién Vive?

3.1 Captura y Variables

Ahora me parece que es el momento para un programa realmente complicado. Aquí está:

```
print "Alto!"
s = raw_input("Quien vive? ")
print "Puede pasar,", s
```

Cuando yo corrí este programa, esto es lo que mostró mi pantalla:

```
Alto!
Quien vive? Josh
Puede pasar, Josh
```

Por supuesto que cuando usted corra el programa, lo que su pantalla muestre será diferente debido a la declaración `raw_input` . Cuando usted corrió el programa probablemente notó (sí corrió el programa, ¿verdad?) que tuvo que escribir su nombre y luego oprimir "Enter". Entonces el programa imprimió un poco más de texto y su nombre. Este es un ejemplo de captura. El programa llega a un punto y espera a que el usuario capture algunos datos que el programa pueda usar después.

Claro que obtener información del usario sería inútil si no tuviéramos donde poner esa información y ahí es donde entran las variables. En el programa precedente `s` es una variable. Las variables son como cajas donde podemos colocar piezas de información. Este es un programa para mostrar ejemplos de variables:

```
a = 123.4
b23 = 'Spam'
nombre = "Bill"
b = 432
c = a + b
print "a + b es", c
print "Nombre de pila es", nombre
print "Partes acomodadas, Despues de Medianoche o",b23
```

Y esta es la salida:

```
a + b es 555.4
Nombre de pila es Bill
Sorted Parts, After Midnight or Spam
```

Las variables almacenan datos. Las variables en el programa anterior son `a`, `b23`, `nombre`, `b`, y `c`. Los dos tipos básicos son hileras y números. Las hileras son una secuencia de letras, números y otros caracteres. En este ejemplo, `b23` y `nombre` son variables que almacenan hileras. `Spam`, `Bill`, `a + b es`, y `Nombre de pila es` son las hileras en este programa. Los caracteres van flanqueados por `"` o `'`. El otro tipo de variable es numérica.

Muy bien, tenemos estas cajas llamadas variables y datos que podemos almacenar en cada variable. La computadora verá un renglón como `nombre = "Bill"` y lo lee como `Pon la hilera Bill dentro de la caja (o variable) nombre.` Después ve la declaración `c = a + b` y la lee como `Pon a + b o 123.4 + 432 o 555.4 dentro de c.`

Este es otro ejemplo de uso de una variable:

```
a = 1
print a
a = a + 1
print a
a = a * 2
print a
```

Y esto es lo que aparece en pantalla:

```
1
2
4
```

Aún si es la misma variable a ambos lados del signo = la computadora lo lee como: `Primero encuentra los datos a almacenar y luego ve dónde van.`

Un programa más antes de que termine este capítulo:

```
num = input("Escriba un numero: ")
str = raw_input("Escriba una hilera: ")
print "num =", num
print "num is a ",type(num)
print "num * 2 =",num*2
print "str =", str
print "str is a ",type(str)
print "str * 2 =",str*2
```

La salida será como sigue:

```
Escriba un numero: 12.34
Escriba una hilera: Hola
num = 12.34
num is a  <type 'float'>
num * 2 = 24.68
str = Hola
str is a  <type 'string'>
str * 2 = HolaHola
```

Note que `num` fue leído por la computadora con `input` mientras que `str` fue leído con `raw_input`. `raw_input` devuelve una hilera mientras que `input` devuelve un número. Cuando quiera el usuario escriba un número use `input` , pero si quiere que el usuario escriba una hilera use `raw_input`.

La segunda mitad del programa usa `type` el cual dice lo que es una variable. Los números son de tipo `int` o `float` (que son formas abreviadas para 'entero' y 'punto flotante' respectivamente). Las hileras son de tipo `string`. Los enteros y los números de punto flotante pueden ser sometidos a operaciones matemáticas, lo que no es posible con hileras. Note cómo cuando python multiplica un número por un entero sucede lo que uno espera. Sin embargo, cuando una hilera es multiplicada por un entero obtenemos una hilera con ese número de copias de la hilera original, p. ej., `str * 2 = HelloHello`.

3.2 Ejemplos

tiempos_y_velocidades.py

```
#Este programa calcula problemas de velocidad y distancia
print "Escriba una velocidad y una distancia"
velocidad = input("Velocidad:")
distancia = input("Distancia:")
print "Tiempo:",distancia/velocidad
```

Corridas de muestra:

```
> python tiempos_y_velocidades.py
Escriba una velocidad y una distancia
Velocidad:5
Distancia:10
Tiempo: 2
> python tiempos_y_velocidades.py
Escriba una velocidad y una distancia
Velocidad:3.52
Distancia:45.6
Tiempo: 12.9545454545
```

Area.py

```
#Este programa calcula el perimetro y el area de un rectangulo
print "Calcula la informacion de un rectangulo"
largo = input("Largo:")
anchuo = input("Ancho:")
print "Area",largo*ancho
print "Perimetro",2*largo+2*ancho
```

Corridas de muestra:

```
> python area.py
Calcula la informacion de un rectangulo
Largo:4
Ancho:3
Area 12
Perimetro 14
> python area.py
Calcula la informacion de un rectangulo
Largo:2.53
Ancho:5.2
Area 13.156
Perimetro 15.46
```

temperatura.py

```
#Convierte Fahrenheit a Celsius
temp = input("Temperatura Farenheit:")
print (temp-32.0)*5.0/9.0
```

Ejemplo de corrida:

```
> python temperatura.py
Temperatura Farenheit:32
0.0
> python temperatura.py
Temperatura Farenheit:-40
-40.0
> python temperatura.py
Temperatura Farenheit:212
100.0
> python temperatura.py
Temperatura Farenheit:98.6
37.0
```

3.3 Ejercicios

Escriba un programa que obtenga 2 variables de hilera y 2 variables enteras del usuario, las concatene (ponga juntas sin espacios) y las imprima en la pantalla, luego multiplique los dos números en un nuevo renglón

4. Cuente hasta 10

4.1 Bucle While

Presentamos nuestra primera estructura de control. Normalmente la computadora empieza con el primer renglón y sigue hacia abajo. Las estructuras de control cambian el orden de ejecución de las declaraciones o deciden si una cierta declaración será ejecutada. Vea a continuación el código para un programa que utiliza la estructura de control while:

```
a = 0
while a < 10:
        a = a + 1
        print a
```

Y esta es la emocionante respuesta del programa:

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

(¿Acaso pensó que transformar su computadora en una calculadora de cinco dólares era lo peor que podía pasar?) ¿Qué hace este programa? Primero ve la declaración `a = 0` y hace `a` igual a cero. Luego ve `while a < 10:` y la computadora revisa si `a < 10`. La primera vez la computadora ve que `a` es igual a cero así que es menor a 10. En otras palabras, mientras a sea menor a 10 la computadora ejecutará las declaraciones con sangría.

He aquí otro ejemplo del uso de while:

```
a = 1
s = 0
print 'Escriba números para añadir a la suma.'
print 'Escriba 0 para salir del programa.'
while a != 0 :
        print 'Suma en este momento:',s
        a = input('Numero? ')
        s = s + a
print 'Suma Total =',s
```

La primera vez que ejecuté este programa Python imprimió en la pantalla:

```
  File "suma.py", line 3
    while a != 0
                ^
SyntaxError: invalid syntax
```

Había olvidado poner : despues del `while`. El mensaje de error indicaba el problema y dónde pensaba que estaba el error. Después de arreglar el problema esto es lo que hice con el programa:

```
Escriba números para añadir a la suma.
Escriba 0 para salir del programa.
Suma en este momento: 0
Numero? 200
Suma en este momento: 200
Numero? -15.25
Suma en este momento: 184.75
Numero? -151.85
Suma en este momento: 32.9
Numero? 10.00
Suma en este momento: 42.9
Numero? 0
Suma Total = 42.9
```

Note cómo `print 'Suma Total =',s` solamente es ejecutado al final. La declaración `while` afecta solamente los renglones con sangría (con margen izquierdo más al centro de la página). El símbolo `!=` significa "no es igual" así que while `a != 0 :` significa "ejecuta las declaraciones con sangría hasta que el valor de a sea cero".

Ahora que tenemos bucles while, es posible tener programas en ejecución para siempre. Un modo muy fácil de hacerlo es escribiendo un programa como siguiente:

```
while 1 == 1:
     print "Auxilio, estoy atrapado en un rizo."
```

Este programa imprimirá en la pantalla `Auxilio, estoy atrapado en un rizo.` hasta la muerte térmica del universo o hasta que usted mismo lo pare. El modo para detenerlo es oprimir las teclas Control (or Ctrl) y `c' (la letra) simultáneamente. Esto matará el programa (Nota: a veces será necesario usar la tecla Enter después de Control-C.)

4.2 Ejemplos

Fibonnacci.py

```
#Este programa calcula la serie de Fibonacci
a = 0
b = 1
conteo = 0
conteo_max = 20
while conteo < conteo_max:
    conteo = conteo + 1
    #Necesitamos guardar 'a' desde que la cambiamos
    a_anterior = a
    b_anterior = b
    a = b_anterior
    b = a_anterior + b_anterior
    # Note que la , al final de la declaracion print evita
    # que comience un nuevo renglon
    print a_anterior,
print
```

Respuesta del programa:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Claveacceso.py

```
# Espera a que capture la clave de acceso.  Use control-C para salir del programa
# sin la clave de acceso

# Note que esta no debe ser la clave de acceso de modo que
# el rizo while sea ejecutado por lo menos una vez.
clave = "fubar"

#note que != significa 'no es igual'
while clave != "unicornio":
    password = raw_input("Clave:")
print "Bienvenido"
```

Muestra:

```
Clave:auo
Clave:y22
Clave:password
Clave:open sesame
Clave:unicornio
Bienvenido
```

5. Decisiones

5.1 Enunciado If

Como siempre, pienso que debo comenzar cada capítulo con un ejercicio de calentamiento así que aquí tiene un programa corto para calcular el valor absoluto de un número:

```
n = input("Numero? ")
if n < 0:
        print "El valor absoluto de",n,"es",-n
else:
        print "El valor absoluto de",n,"es",n
```

A continuación, vea el resultado en pantalla de las dos veces que ejecuté el programa:

```
Numero? -34
El valor absoluto de -34 es 34

Numero? 1
El valor absoluto de 1 es 1
```

¿Qué hace la computadora cuando ve este código? Primero, solicita un número al usuario con la declaración `n = input("Numero? ")`. Luego lee el renglón `if n < 0:` Si `n` es menor a cero, Python ejecuta el renglón `print "El valor absoluto de ",n,"es",-n`. De otro modo, Python ejecuta el renglón `print "El valor absoluto de",n,"es",n`.

De un modo más formal, Python evalúa si la expresión `n < 0` es verdadera o falsa. Una declaración `if` es seguida por un bloque de enunciados que son ejecutados cuando la expresión es verdadera. Opcionalmente, después del `if` puede haber un enunciado `else`. El enunciado `else` es ejecutado si la expresión evaluada en el `if` resulta falsa.

Una expresión puede ser evaluada de varios modos. He aquí una tabla de todos ellos:

operador	*función*
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
=	Igual a
!=	No es igual a
<>	Otro modo para “No es igual a”

Otro aspecto del comando `if` es la declaración `elif`. Es una combinación de "`else if`" y significa "si el if original es falso y la expresión de `elif` es verdadera haz esa parte". Un ejemplo:

```
a = 0
while a < 10:
        a = a + 1
        if a > 5:
                print a," > ",5
        elif a <= 7:
                print a," <= ",7
        else:
                print "Ninguna expresion fue verdadera"
```

y la ejecución del programa

```
1  <=  7
2  <=  7
3  <=  7
4  <=  7
5  <=  7
6  >  5
7  >  5
8  >  5
9  >  5
10  >  5
```

Note cómo `elif a <= 7` solamente es evaluado cuando la declaración `if` resulta falsa. `elif` permite varias evaluaciones en una sola declaración `if`.

5.2 Ejemplos

Alto_bajo.py

```
# Juega adivinar el numero por alto o bajo
# (originalmente escrito por Josh Cogliati, mejorado por Quique)

# Esto deberia ser semi-aleatorio, como los dos ultimos digitos de
# la hora o algo asi, pero eso tendra que esperar
# a un capitulo posterior.  (Puntos extra, modifique el programa
# para hacerlo aleatorio despues del capitulo "Modulos")
numero = 78
intento = 0

while intento != numero :
    intento = input ("Trata de adivinar mi numero: ")

    if intento > numero :
        print "Muy alto"

    elif intento < numero :
            print "Muy bajo"

print "Exacto"
```

Muestra:

```
Trata de adivinar mi numero:100
Muy alto
Trata de adivinar mi numero:50
Muy bajo
Trata de adivinar mi numero:75
Muy bajo
```

```
Trata de adivinar mi numero:87
Muy alto
Trata de adivinar mi numero:81
Muy alto
Trata de adivinar mi numero:78
Exacto
```

par.py

```
#Pide un numero.
#Imprime si es par o non

numero = input("Dime un numero: ")
if numero % 2 == 0:
    print numero,"es par."
elif numero % 2 == 1:
    print number,"es non."
else:
    print number,"es muy raro."
```

Muestra:

```
Dime un numero: 3
3 es non.

Dime un numero: 2
2 es par.

Dime un numero: 3.14159
3.14159 es muy raro.
```

5.3 Ejercicios

Modifique el programa de clave de acceso para llevar nota del número de intentos equivocados. Si es más de tres veces, imprima ``Debió ser muy complicado."

Escriba un programa que pida dos números. Si la suma de los números es mayor que 100, imprima `print ``Que numerotes!''`.

Escriba un programa que pida el nombre del usuario, si escriben su nombre (el de usted, autor del programa) diga "Es un nombre muy bonito", si escriben "John Cleese" o "Michael Palin", dígales lo que piensa de ellos ;), de otro modo diga "Tienes un nombre agradable".

6. Depuración

6.1 ¿Qué es la depuración?

> *Tan pronto como empezamos a programar, encontramos para nuestra sorpresa que obtener programas correctos no sería tan sencillo como habíamos pensado. Faltaba por descubrir la depuración. Puedo recordar el instante exacto cuando me di cuenta que una gran parte de mi vida a partir de ese momento sería empleada en encontrar errores en mis propios programas.*
>
> *- Maurice Wilkes descubre la depuración, 1949*

En este punto, si ha estado jugando con los programas, probablemente haya visto que a veces el programa hace cosas que usted no quería que hiciera. Esto es muy común. Depurar es el proceso de descubrir lo que la computadora está haciendo y hacerla que haga lo que usted quiere. Esto puede ser muy difícil. Una vez tardé una semana rastreando un error causado por alguien que puso una `x` donde debió haber una `y`.

Este capítulo será más abstracto que los anteriores. Díganme si es útil.

6.2 ¿Qué debería hacer el programa?

Lo primero que hay que hacer (parece muy obvio) es aclarar lo que el programa debería hacer si funcionara correctamente. Invente algunos casos de prueba y vea lo que sucede. Por ejemplo, digamos que tengo un programa que calcula el perímetro de un rectángulo (la suma de todos sus lados). Tengo los siguientes casos de prueba:

anchura	*altura*	*perímetro*
3	4	14
2	3	10
4	4	16
2	2	8
5	1	12

Ahora ejecuto mi programa para todos los casos de prueba y veo si el programa hace lo que espero que haga. Si no lo hace, necesito hallar lo que la computadora esta haciendo.

Por lo común, algunos de los casos de prueba funcionarán bien y otros no. Si este es su caso, trate de descubrir lo que tienen en común los casos que sí funcionan. Por ejemplo, esta es la respuesta de un programa para calcular el perímetro (verá el código en un minuto):

```
Altura: 3
Anchura: 4
Perimetro =  15

Altura: 2
Anchura: 3
Perimetro =  11
```

```
Altura: 4
Anchura: 4
Perimetro =  16

Altura: 2
Anchura: 2
Perimetro =  8

Altura: 5
Anchura: 1
Perimetro =  8
```

Note que no funcionó para los dos primeros casos, funcionó bien en los siguientes dos y falló en el último. Trate de descubrir lo que tienen en común los casos que funcionaron. Una vez que tenga idea del problema, encontrar la causa será más sencillo. Si fuera necesario, trate más casos de prueba con sus programas.

6.3 ¿Qué hace realmente el programa?

El siguiente paso es ver el código fuente. Una de las cosas más importantes que uno hace cuando programa es leer código. El modo principal de hacer esto es hacer un seguimiento del código.

Un seguimiento del código empieza en el primer renglón, y sigue hasta que el programa termina. El bucle `while` y las declaraciones `if` pueden causar que algunos renglones nunca sean ejecutados, mientras que otros sean ejecutados muchas veces. En cada renglón usted descubre lo que Python ha hecho.

Empecemos con el simple programa del perímetro. No lo escriba en el interpretador, vamos a leerlo, no a ejecutarlo. El código fuente es:

```
altura = input("Altura: ")
anchura = input("Anchura: ")
print "Perimetro = ",anchura+altura+anchura+anchura
```

Pregunta: ¿Cuál es el primer renglón que Python ejecuta?

Respuesta: El primer renglón es ejecutado primero, siempre. En este caso, es: `altura = input("Altura: ")`

Pregunta: ¿Qué hace ese renglón?

Respuesta: Imprime `Altura:` , espera que el usuario escriba un número y coloca ese valor en la variable altura.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: En general, es el siguiente renglón, en este caso: `anchura = input("Anchura: ")`

Pregunta: ¿Qué hace ese renglón?

Respuesta: Imprime `Anchura:` , espera que el usuario escriba un número y coloca ese número en la variable anchura.

Pregunta: ¿Cuál es el siguiente renglón ejecutado?

Respuesta: Cuando el siguiente renglón carece de sangría, es el renglón que sigue inmediatamente, así que es: `print "Perimetro = ",anchura+altura+anchura+anchura` (Es posible ejecutar una función, pero eso le toca a otro capítulo.)

Pregunta: ¿Qué hace ese renglón?

Respuesta: Primero imprime `perimetro =`, y luego imprime el valor que resulta de `anchura+altura+anchura+anchura`

Pregunta: ¿`anchura+altura+anchura+anchura` constituye un cálculo correcto del perímetro?

Respuesta: Veamos, el perímetro de un rectángulo es el fondo (anchura) más el lado izquierdo (altura) más el lado superior (anchura) más el lado derecho (altura) (¿eh?). El último valor debería ser la longitud del lado derecho, no la altura.

Pregunta: ¿Comprende porqué algunas veces el perímetro fue calculado "correctamente"?

Respuesta: El resultado era correcto cuando la anchura y la altura eran iguales.

El siguiente programa a través del cual caminaremos es un programa que debería imprimir 5 puntos en la pantalla. Sin embargo, esta es la salida del programa:

```
. . . .
```

Y este es el programa:

```
numero = 5
while numero > 1:
    print ".",
    numero = numero - 1
print
```

El seguimiento en este programa será más complicado, dado que ahora hay porciones con sangría (o estructuras de control). Empecemos:

Pregunta: ¿Cuál es el primer renglón que Python ejecuta?

Respuesta: El primer renglón del archivo: `numero = 5`

Pregunta: ¿Qué hace ese renglón?

Respuesta: Coloca en número 5 en la variable numero.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: El siguiente renglón es: `while numero > 1:`

Pregunta: ¿Qué hace ese renglón?

Respuesta: Las declaracion `while` , en general, evalúan la expresión y si es verdadera ejecuta el

bloque de código sangrado, de otro modo salta el bloque sin ejecutarlo.

Pregunta: ¿Qué hace ahora?

Respuesta: Si `numero > 1` es verdadero, los dos siguientes renglones serán ejecutados.

Pregunta: Es `numero > 1`?

Respuesta: El último valor colocado en numero fue 5 y 5 > 1 así que la respuesta es sí.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: Dado que el `while` fue verdadero, el siguiente renglón es: `print "."`,

Pregunta: ¿Qué hace ese renglón?

Respuesta: Imprime un punto y como el enunciado termina con una `","`, la siguiente impresión será en el mismo renglón de la pantalla.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: `numero = numero - 1` ya que es el siguiente renglón y no hay cambios de sangría.

Pregunta: ¿Qué hace ese renglón?

Respuesta: Calcula `numero - 1`, el cual es el valor presente de numero (o 5) le resta 1 y coloca el nuevo valor en numero. Básicamente, cambia el valor de numero de 5 a 4.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: El nivel de sangría disminuye, así que tenemos que ver qué tipo de estructura de control es ésta. Es un rizo while, así que regresamos a la cláusula `while` que `dice while numero > 1:`

Pregunta: ¿Qué hace ese renglón?

Respuesta: Evalúa numero, resulta igual a 4, y lo compara con 1. Ya que 4 > 1 el rizo while continúa.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: Como el rizo while fue verdadero, el siguiente renglón es: `print "."`,

Pregunta: ¿Qué hace ese renglón?

Respuesta: Imprime un segundo punto en el renglón.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: No hay cambio de sangría, así que es: `number = number - 1`

Pregunta: ¿Y qué hace ese renglón?

Respuesta: Toma el valor presente de numero (4), le resta 1, lo que resulta en 3 y coloca 3 en numero.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: Gracias al cambio de sangría, el siguiente renglón es: while numero > 1:

Pregunta: ¿Qué hace ese renglón?

Respuesta: Compara el valor presente de numero (3) a 1. 3 > 1 así que el rizo while continúa.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: Como lea condición del rizo while fue verdadera, el siguiente renglón es:`print "."`,

Pregunta: ¿Y qué hace?

Respuesta: Imprime un tercer punto en el renglón de la pantalla.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: Es: `numero = numero - 1`

Pregunta: ¿Qué hace ese renglón?

Respuesta: Toma el valor presente de numero (3), le resta 1 y hace 2 el nuevo valor de numero.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: Regresa al principio del rizo while: `while numero > 1:`

Pregunta: ¿Qué hace ese renglón?

Respuesta: Compara el valor presente de numero (2) con 1. Dado que 2 > 1 el rizo while continúa.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: Como el rizo while continúa: `print "."`,

Pregunta: ¿Qué hace ese renglón?

Respuesta: Descubre el significado de la vida, el universo y todo. Es una broma. (Sólo quería asegurarme de que estuviera poniendo atención.) Imprime un cuarto punto en la pantalla.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: Esto es: `numero = numero - 1`

Pregunta: ¿Qué hace ese renglón?

Respuesta: Toma el valor presente de numero (2) le resta 1 y hace 1 el nuevo valor de numero.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: Regresa al principio del rizo while: **while numero > 1:**

Pregunta: ¿Qué hace ese renglón?

Respuesta: Compara el valor presente de numero (1) con 1. Como 1 > 1 es falso (uno no es mayor que uno), el rizo while termina.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: Ya que la condición del rizo look fue falsa, el siguiente renglón es aquél después de que el rizo sale, o: print

Pregunta: ¿Qué hace ese renglón?

Respuesta: Hace que la pantalla vaya al siguiente renglón.

Pregunta: ¿Porqué el programa no imprime 5 puntos?

Respuesta: El rizo sale antes del último punto.

Pregunta: ¿Cómo podemos arreglar eso?

Respuesta: Hacemos que el rizo salga después de imprimir un punto más.

Pregunta: Y cómo hacemos eso?

Respuesta: Hay varios modos. Uno sería cambiar la condición del rizo while a: `while numero > 0:` Otro modo sería cambiar el enunciado condicional a: `numero >= 1` Existen otros modos más.

6.4 ¿Cómo arreglo el programa?

Necesita descubrir lo que el programa está haciendo. Necesita definir lo que el programa debería hacer. Encuentre cuál es la diferencia entre los dos. Depurar es una habilidad aprendida con la práctica. Si no puede descubrir el problema después de aproximadamente una hora, tómese un descanso, hable con alguien acerca del problema y piense en la imortalidad del cangrejo. Regrese después de un rato y probablemente tendrá nuevas ideas acerca del problema. Buena suerte.

7. Definiendo Funciones

7.1 Creando Funciones

Para arrancar este capítulo, le voy a dar un ejemplo de lo pudiera pero no debiera hacer (así que no lo escriba):

```
a = 23
b = -23

if a < 0:
    a = -a

if b < 0:
    b = -b

if a == b:
    print "Los valores absolutos de", a,"y",b,"son iguales"
else:
    print "Los valores absolutos de a y b son diferentes"
```

y la salida del programa es:

```
Los valores absolutos de 23 y 23 son iguales
```

Este programa parece un poco repetitivo. (Los programadores odian repetir cosas (Para eso están las computadoras, ¿verdad?)) Afortunadamente, Python permite crear funciones para eliminar duplicación. Aquí está el ejemplo reescrito:

```
a = 23
b = -23

def mi_abs(num):
    if num < 0:
        num = -num
    return num

if mi_abs(a) == mi_abs(b):
    print "Los valores absolutos de", a,"y",b,"son iguales"
else:
    print "Los valores absolutos de a y b son diferentes"
```

y la salida del programa es:

```
Los valores absolutos de 23 y -23 son iguales
```

El aspecto clave de este programa es el enunciado `def`. `def` (apócope de define) comienza una definición de función. A `def` le sigue el nombre de la función `mi_abs`. Luego viene un `(` seguido de un parámetro `num` (el valor de `num` es pasado del programa a la función cuando la función es llamada). Los enunciados después de `:` son ejecutados cuando usamos la función. Los enunciados continúan hasta que los enunciados sangrados terminan, o hasta que encuentra un `return`. El enunciado `return` devuelve un valor al lugar donde la función fue llamada.

Note cómo los valores de `a` y `b` no sufren alteración. Por supuesto que podemos usar funciones para repetir tareas que no devuelven un valor. Unos ejemplos:

```
def hola():
    print "Hola"

def area(anchura,altura):
    return anchura*altura

def imprime_bienvenida(nombre):
    print "Bienvenido",nombre

hola()
hola()

imprime_bienvenida("Pedro")
ancho = 4
alto = 5
print "anchura =",ancho,"altura =",alto,"area =",area(ancho,alto)
```

y la salida es:

```
Hola
Hola
Bienvenido Pedro
anchura = 4 altura = 5 area = 20
```

Ese ejemplo muestra un poco más de lo que puede hacer con funciones. Note que puede usar cero argumentos, o dos o más. Note también que cuando una función no devuelve un valor, el return es opcional.

7.2 Variables en funciones

Por supuesto, cuando uno elimina código repetido, con frecuencia hay variables en ese código. Python resuelve ese problema de un modo especial. Hasta ahora, todas las variables que hemos visto son variables globales. Las funciones tienen un tipo especial de variable llamada variable local. Estas variables sólo existen mientras la función está en ejecución. Cuando una variable local tiene el mismo nombre que una variable global, la variable local esconde a la variable global. ¿Confundido? Bueno, espero que el siguiente ejemplo (el cual es un poco complicado) aclare las cosas.

```
var_a = 10
var_b = 15
var_e = 25

def func_a(var_a):
    print "in func_a var_a = ",var_a
    var_b = 100 + var_a
    var_d = 2*var_a
    print "en func_a var_b = ",var_b
    print "en func_a var_d = ",var_d
    print "en func_a var_e = ",var_e
    return var_b + 10

var_c = func_a(var_b)
```

```
print "var_a = ",var_a
print "var_b = ",var_b
print "var_c = ",var_c
print "var_d = ",var_d
```

La salida es:

```
en func_a var_a =  15
en func_a var_b =  115
en func_a var_d =  30
en func_a var_e =  25
var_a =  10
var_b =  15
var_c =  125
var_d =
Traceback (innermost last):
  File "separate.py", line 20, in ?
    print "var_d = ",var_d
NameError: var_d
```

En este ejemplo, las variables `var_a`, `var_b`, y `var_d` son variables locales cuando están dentro de la función `func_a`. Después de ejecutar el enunciado `return var_b + 10`, dejan de existir. La variable `var_a` es automáticamente una variable local ya que es un nombre de parámetro. Las variables `var_b` y `var_d` son variables locales dado que aparecen a la izquierda de un signo igual en la función en las declaraciones `var_b = 100 + var_a` y `var_d = 2*var_a`

Como puede ver, una vez que la función termina, las variables locales `var_a` y `var_b` que habían escondido las variables globales del mismo nombre desaparecen. Entonces el enunciado `print "var_a = ",var_a` imprime el valor 10 y no 15 ya que la variable local que escondía la variable global ya no está.

Otra cosa que hay que notar es el `NameError` que sucede al final. Esto aparece porque la variable `var_d` ya no existe desde que `func_a` terminó. Todas las variables locales son borradas cuando la función sale. Si quiere que la función proporcione algo, tendrá que usar `return` algo.

Un último punto a notar es que el valor de `var_e` permanece sin cambiar dentro de `func_a` ya que no es un parámetro y nunca aparece a la izquierda de un signo igual dentro de `func_a`. Cuando una variable global es leída dentro de una función, es la variable global de afuera de la función.

Las funciones permiten variables locales que existen sólo dentro de la función y que pueden ocultar otras variables que están fuera de la función.

7.3 Seguimiento de la función

Ahora sigamos el siguiente programa:

```
def mult(a,b):
    if b == 0:
        return 0
    rest = mult(a,b - 1)
    valor = a + rest
    return valor

print "3*2 = ",mult(3,2)
```

Basicamente este programa crea una función para multiplicar enteros positivos (que es mucho más lenta que la función interconstruida correspondiente de python) y luego demuestra esta función con el uso de la función.

Pregunta: ¿Qué es lo primero que hace el programa?

Respuesta: Lo primero que hace el programa es definir la función mult con los siguientes renglones:

```
def mult(a,b):
    if b == 0:
        return 0
    rest = mult(a,b - 1)
    valor = a + rest
    return valor
```

Esto crea una función que toma dos parámetros y devuelve un valor cuando ha terminado. Esta función puede ser ejecutads después.

Pregunta: ¿Luego qué pasa?

Respuesta: El siguiente renglón después de la función, `print "3*2 = ",mult(3,2)` es ejecutado.

Pregunta: ¿Y esto que hace?

Respuesta: Imprime 3*2 = y el valor devuelto por `mult(3,2)`

Pregunta: ¿Y qué devuelve `mult(3,2)`?

Respuesta: Necesitamos caminar a través de la función `mult` para descubrirlo.

Pregunta: ¿Luego qué pasa?

Respuesta: La variable `a` es asignada el valor 3 y la variable `b` es asignada el valor 2.

Pregunta: ¿Y luego?

Respuesta: La declaración `if b == 0:` es ejecutada. Ya que el valor de `b` es igual a 2, esta expresión resulta falsa así que nos saltamos el `return 0`.

Pregunta: ¿Y luego qué?

Respuesta: El renglón `rest = mult(a,b - 1)` es ejecutado. Este renglón asigna a la variable local `rest` el valor de `mult(a,b - 1)`. El valor de `a` es 3 y el valor de `b` es 2 así que la función llamada es `mult(3,1)`

Pregunta: ¿Entonces cuál es el valor de `mult(3,1)` ?

Respuesta: Necesitamos ejecutar la función `mult` con los parámetros 3 y 1.

Pregunta: ¿Luego qué pasa?

Respuesta: Las variables locales en la más reciente llamada de la función son asignadas valores, así que `a` es 3 y `b` es ahora 1. Ya que estas son variables locales, no afectan los valores previos de `a` y `b`.

Pregunta: ¿Y luego?

Respuesta: Dado que el valor de `b` es 1, la expresión evaluada por el `if` es falsa, el siguiente renglón ejecutado es `rest = mult(a,b - 1)`.

Pregunta: ¿Y este renglón qué hace?

Respuesta: Este renglón asigna el valor de `mult(3,0)` a `rest`.

Pregunta: ¿Cuál es ese valor?

Respuesta: Tendremos que ejecutar la función una vez más para saber. Esta vez `a` vale 3 y `b` vale 0.

Pregunta: ¿Luego qué pasa?

Respuesta: El primer renglón en la función es `if b == 0:`. `b` vale 0 así que el siguiente renglón ejecutado es `return 0`

Pregunta: ¿Y qué hace `return 0`?

Respuesta: Este renglón devuelve el valor 0 hacia afuera de la función.

Pregunta: ¿Y?

Respuesta: Así que ahora sabemos que `mult(3,0)` vale 0. Ahora sabemos lo que hizo `rest = mult(a,b - 1)` ya que ejecutamos la función `mult` con los parámetros 3 y 0. Hemos terminado ejecutando `mult(3,0)` y estamos de vuelta ejecutando `mult(3,1)`. La variable `rest` obtiene el valor 0.

Pregunta: ¿Cuál es el siguiente renglón?

Respuesta: El renglón `valor = a + rest` es ejecutado a continuación. En esta ejecución, `a=3` y `rest=0` así que `valor=3`.

Pregunta: ¿Luego qué pasa?

Respuesta: El renglón `return` valor es ejecutado. Esto devuelve 3 desde la función. Esto también sale de ejecutar la función `mult(3,1)`. Después de ejecutar `return`, regresamos a la ejecución de `mult(3,2)`.

Pregunta: ¿Dónde nos quedamos en `mult(3,2)`?

Respuesta: Teníamos las variables `a=3` y `b=2` y estábamos examinando el renglón `rest = mult(a,b - 1)`.

Pregunta: ¿Y ahora qué pasa?

Respuesta: La variable `rest` es asignada el valor 3. El siguiente renglón `value = a + rest` le asigna a valor el resultado de 3+3 o 6.

Pregunta: ¿Y luego qué pasa?

Respuesta: Es ejecutado el siguiente renglón, éste devuelve 6 desde la función. Ahora estamos de vuelta ejecutando `print "3*2 = ",mult(3,2)` el cual puede ahora imprimir el 6.

Pregunta: ¿Qué es lo que ha sucedido globalmente?

Respuesta: Básicamente usamos dos hechos para multiplicar dos números. El primero es que cualquier número multiplicado por 0 es 0 (x * 0 = 0). El segundo es que un número multiplicado por otro es igual al primer número más el primer número multiplicado por el segundo número menos uno (x * y = x + x * (y - 1)). Así que lo que sucede es que 3*2 es convertido a 3 + 3*1. Luego 3*1 es convertido a 3 + 3*0. Sabemos que cualquier número por cero es cero entonces 3*0 es 0. Luego podemos calcular que 3 + 3*0 es 3 + 0 el cual es 3. Ahora sabemos el valor de 3*1 y podemos calcular que 3 + 3*1 es 3 + 3 el cual es 6.

Así es como funciona esto:

```
3*2
3 + 3*1
3 + 3 + 3*0
3 + 3 + 0
3 + 3
6
```

Estas dos secciones fueron escritas recientemente. Si tiene algún comentario, ha encontrado errores o piensa que necesito más o mejores explicaciones envíeme un correo electrónico. En el pasado, he logrado hacer que cosas muy simples sean completamente incomprensibles. Si el resto del libro tiene sentido pero esta sección no lo tiene, probablemente es por un error mío y me gustaría saber. Gracias.

7.4 Ejemplos

`factorial.py`

```
#define una funcion que calcula el factorial

def factorial(n):
    if n <= 1:
        return 1
    return n*factorial(n-1)

print "2! = ",factorial(2)
print "3! = ",factorial(3)
print "4! = ",factorial(4)
print "5! = ",factorial(5)
```

Salida:

```
2! =  2
3! =  6
4! =  24
5! =  120
```

temperatura2.py

```
#convierte temperatura a fahrenheit o celsius

def imprime_opciones():
    print "Opciones:"
    print " 'p' imprime opciones"
    print " 'c' convierte de celsius"
    print " 'f' convierte de fahrenheit"
    print " 'q' termina el programa"

def celsius_a_fahrenheit(c_temp):
    return 9.0/5.0*c_temp+32

def fahrenheit_a_celsius(f_temp):
    return (f_temp - 32.0)*5.0/9.0

seleccion = "p"
while seleccion != "q":
    if seleccion == "c":
        temp = input("Temperatura en Celsius:")
        print "Fahrenheit:",celsius_to_fahrenheit(temp)
    elif seleccion == "f":
        temp = input("Temperatura en Fahrenheit:")
        print "Celsius:",fahrenheit_to_celsius(temp)
    elif seleccion != "q":
        imprime_opciones()
    seleccion = raw_input("opcion:")
```

Muestra:

```
> python temperatura2.py
Opciones:
 'p' imprime opciones
 'c' convierte de celsius
 'f' convierte de fahrenheit
 'q' termina el programa
opcion:c
Temperatura en Celsius:30
Fahrenheit: 86.0
opcion:f
Temperatura en Fahrenheit:60
Celsius: 15.5555555556
opcion:q
```

area2.py

```
#By Amos Satterlee
#traducido por Victor M. Rosas-Garcia
print
def hola():
    print 'Hola!'

def area(anchura,altura):
    return anchura*altura

def imprime_bienvenido(nombre):
    print 'Bienvenido,',nombre

name = raw_input('Su nombre: ')
hola(),
imprime_bienvenido(nombre)
print
print 'Para hallar el area de un rectangulo,'
print 'Escriba la anchura y la altura abajo.'
print
an = input('Anchura:  ')
while an <= 0:
    print 'Debe ser un número positivo'
    an = input('Anchura:  ')
al = input('altura: ')
while al <= 0:
    print 'Debe ser un número positivo'
    al = input('Altura: ')
print 'Anchura =',an,' Altura =',al,' entonces el Area =',area(al,an)
```

Muestra:

```
Su nombre: Josh
Hola!
Bienvenido, Josh

Para hallar el area de un rectangulo,
Escriba la anchura y la altura abajo.

Anchura:  -4
Debe ser un número positivo
Anchura:  4
Altura: 3
Anchura = 4  Altura = 3  entonces el Area = 12
```

7.5 Ejercicios

Vuelva a escribir el programa area.py de la sección 3.2 para tener funciones separadas para el área de un cuadrado, el área de un rectángulo y el área de un círculo. (3.14 * radio**2). Este programa debería incluir una interfase de menú.

8. Listas

8.1 Variables con más de un valor

Ya ha visto que las variables ordinarias almacenan un solo valor. Sin embargo, otros tipos de variable pueden almacenar más de un valor. El tipo más simple es llamado una lista. He aquí un ejemplo de una lista en uso:

```
cual = input("Cual mes (1-12)? ")
meses = ['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio', 'Julio',\
      'Agosto', 'Septiembre', 'Octubre', 'Noviembre', 'Diciembre']
if 1 <= cual <= 12:
        print "El mes es",meses[cual - 1]
```

y un ejemplo de la salida:

```
Cual mes (1-12)? 3
El mes es Marzo
```

En este ejemplo meses es una lista. meses queda definida con los renglones `meses = ['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio', 'Julio',\ 'Agosto', 'Septiembre', 'Octubre', 'Noviembre', 'Diciembre']` (Note que puede usar una \ para dividir un renglón muy largo). `[` y `]` abren y cierran la lista, mientras las comas (``,'') sirven para separar los elementos de la lista. Usamos la lista en `meses[cual - 1]`. Una lista consiste de elementos numerados a partir de ***cero***. En otras palabras, si desea Enero debe referirse a `meses[0]`. Proporcione un número a una lista y la lista le devolverá el valor almacenado en ese lugar.

El enunciado `if 1 <= cual <= 12:` será verdadero si el valor de cual está entre uno y doce, ambos inclusive (en otras palabras, es lo que usted esperaría si ha visto esto en álgebra).

Podemos considerar la listas como series de cajas. Por ejemplo, las cajas creadas por `demolista = ['vida',42, 'el universo', 6,'y',7]` pudieran verse así:

Número de caja	***0***	***1***	***2***	***3***	***4***	***5***
demolista	'vida'	42	'el universo'	6	'y'	7

Nos referimos a cada caja por su número, así que el enunciado `demolista[0]` obtendría 'vida', `demolista[1]` obtendría 42 y así sucesivamente hasta `demolista[5]` que resultaría en 7.

8.2 Más aspectos de las listas

El siguiente ejemplo es sólo para mostrar muchas otras cosas que las listas pueden hacer (sólo por esta vez no espero que lo escriba, pero sí debiera jugar con listas hasta que se sienta cómodo con ellas). Aquí va:

```
demolista = ['vida',42, 'el universo', 6,'y',7]
print 'demolista = ',demolista
demolist.append('todas las cosas')
print "despues de anexar 'todas las cosas' ahora demolista es:"
print demolista
print 'len(demolista) =', len(demolista)
```

```
print 'demolista.index(42) =',demolista.index(42)
print 'demolista[1] =', demolista[1]
#Ahora haremos un rizo a traves de la lista
c = 0
while c < len(demolista):
    print 'demolista[',c,']=',demolista[c]
    c = c + 1
del demolista[2]
print "Despues de remover 'el universo' ahora demolista es:"
print demolista
if 'vida' in demolista:
    print "encontre 'vida' en demolista"
else:
    print "no encontre 'vida' en demolista"
if 'amiba' in demolista:
    print "encontre 'amiba' en demolista"
if 'amiba' not in demolista:
    print "no encontre 'amiba' en demolista"
demolista.sort()
print 'La demolist ordenada es ',demolista
```

La salida es:

```
demolista =  ['vida', 42, 'el universo', 6, 'y', 7]
despues de anexar 'todas las cosas' ahora demolista es:
['vida', 42, 'el universo', 6, 'y', 7, 'todas las cosas']
len(demolista) = 7
demolista.index(42) = 1
demolista[1] = 42
demolista[ 0 ]= vida
demolista[ 1 ]= 42
demolista[ 2 ]= el universo
demolista[ 3 ]= 6
demolista[ 4 ]= y
demolista[ 5 ]= 7
demolista[ 6 ]= todas las cosas
Despues de remover 'el universo' ahora demolista es:
['vida', 42, 6, 'y', 7, 'todas las cosas']
encontre 'vida' en demolista
no encontre 'amiba' en demolista
La demolist ordenada es  [6, 7, 42, 'todas las cosas', 'vida', 'y']
```

Este ejemplo usa un montón de funciones nuevas. Note que puede imprimir toda la lista simplemente con `print`. La función `append` se usa para añadir un elemento nuevo al final de la lista. `len` devuelve cuántos elementos hay en la lista. Los índices válidos de una lista (esto es, los números que pueden usarse dentro de `[]`) varían de `0` a `len - 1`. La función `index` dice cuál es la primera ubicación de un elemento en una lista. Note cómo `demolista.index(42)` devuelve 1 y cuando `demolista[1]` es ejecutado devuelve 42. El renglón `#Ahora haremos un rizo a traves de la lista` es sólo un recordatorio para el programador (también llamado un comentario). Python no hace caso de los renglones que comienzan con `#`. Luego, los renglones:

```
c = 0
while c < len(demolista):
    print 'demolista[',c,']=',demolista[c]
    c = c + 1
```

Crean una variable `c` la cual comienza con el valor 0 y es incrementada hasta que alcanza el último índice de la lista. Mientras tanto, el enunciado `print` imprime cada elemento de la lista.

El comando `del` puede ser usado para remover de la lista un elemento en particular. Los siguientes renglones usan el operador `in` para evaluar si un elemento dado está o no en la lista.

La función `sort` ordena la lista. Esto es útil si necesita una lista ordenada de menor a mayor número o alfabéticamente. Observe que esto reacomoda la lista.

En resumen, para una lista puede uno usar las siguientes operaciones:

ejemplo	*explicación*
`lista[2]`	devuelve el elemento de índice 2
`lista[2] = 3`	asigna el valor 3 al elemento de índice 2
`del lista[2]`	remueve el elemento de índice 2
`len(lista)`	devuelve la longitud de lista
`"valor" in lista`	es verdadero si `"valor"` es un elemento de lista
`"valor" not in lista`	es verdadero si `"valor"` no es un elemento en lista
`lista.sort()`	ordena lista
`lista.index("valor")`	devuelve el índice del primer lugar donde aparece `"valor"`
`lista.append("valor")`	anexa un elemento "valor" al final de lista

El siguiente ejemplo usa estas operaciones de un modo más útil:

```
articulo_menu = 0
lista = []
while articulo_menu != 9:
        print "--------------------"
        print "1. Imprime la lista"
        print "2. Anexa un nombre a la lista"
        print "3. Remueve un nombre de la lista"
        print "4. Cambia un elemento de la lista"
        print "9. Cierra el programa"
        articulo_menu = input("Elija un articulo del menu: ")
        if articulo_menu == 1:
                actual = 0
                if len(lista) > 0:
                        while actual < len(lista):
                                print actual,". ",list[actual]
                                actual = actual + 1
                else:
                        print "la lista esta vacia"
        elif articulo_menu == 2:
                nombre = raw_input("Escriba un nombre para anexar: ")
                lista.append(nombre)
        elif articulo_menu == 3:
                borra_nombre = raw_input("Cual nombre le gustaria quitar: ")
                if borra_nombre in lista:
                        numero_elemento = lista.index(borra_nombre)
                        del lista[numero_elemento]
                        #El codigo anterior solo borra la primera instancia
                        # del nombre.  El siguiente codigo debido a Gerald quita todas.
                        #while borra_nombre in lista:
                        #       numero_elemento = lista.index(borra_nombre)
                        #       del lista[numero_elemento]
```

```
                else:
                        print borra_nombre," no esta en la lista"
        elif articulo_menu == 4:
                nombre_viejo = raw_input("Cual nombre le gustaria cambiar: ")
                if nombre_viejo in lista:
                        numero_articulo = lista.index(nombre_viejo)
                        nombre_nuevo = raw_input("Cual es el nuevo nombre: ")
                        lista[numero_articulo] = nombre_nuevo
                else:
                        print nombre_viejo," no esta en la lista"
print "Adios"
```

Y esta es parte de la salida:

```
--------------------
1. Imprime la lista
2. Anexa un nombre a la lista
3. Remueve un nombre de la lista
4. Cambia un elemento de la lista
9. Cierra el programa

Elija un articulo del menu: 2
Escriba un nombre para anexar: Jack

Elija un articulo del menu: 2
Escriba un nombre para anexar: Jill

Elija un articulo del menu: 1
0 .  Jack
1 .  Jill

Elija un articulo del menu: 3
Cual nombre le gustaria quitar: Jack

Elija un articulo del menu: 4
Cual nombre le gustaria cambiar: Jill
Cual es el nuevo nombre: Jill Peters

Elija un articulo del menu: 1
0 .  Jill Peters

Elija un articulo del menu: 9
Adios
```

Ese fue un programa largo. Veamos el código fuente. El renglón `lista = []` hace la variable lista una lista sin artículos (o elementos). El siguiente renglón importante es `while articulo_menu != 9:`. Este renglón comienza un rizo que habilita el menú de opciones para este programa. Los siguientes renglones muestran el menú y deciden qué parte del programa ejecutar.

La sección:

```
actual = 0
if len(lista) > 0:
        while actual < len(lista):
                print actual,". ",lista[actual]
                actual = actual + 1
else:
        print "La lista esta vacia"
```

recorre la lista e imprime cada nombre. `len(lista)` dice cuántos elementos hay en una lista. Si `len` devuelve 0 entonces la lista está vacía.

Luego, unos cuantos rengloes después, aparece el enunciado `lista.append(nombre)`. Usa la función `append` para anexar un elemento al final de la lista. Salte dos renglones más y observe esta sección del código:

```
numero_articulo = lista.index(borra_nombre)
del lista[numero_articulo]
```

Aquí la función `index` se emplea para encontrar el valor del índice que será usado después para remover el artículo. `del lista[numero_articulo]` se emplea para remover un artículo de la lista.

La siguiente sección

```
nombre_viejo = raw_input("Cual nombre le gustaria cambiar: ")
if nombre_viejo in lista:
        numero_articulo = lista.index(nombre_viejo)
        nombre_nuevo = raw_input("Cual es el nuevo nombre: ")
        lista[numero_articulo] = nombre_nuevo
else:
        print nombre_viejo," no esta en la lista"
```

usa `index` para hallar el `numero_articulo` y luego pone `nombre_nuevo` en lugar de `nombre_viejo`.

8.3 Ejemplos

prueba.py

```
## Este programa hace una prueba de conocimientos

verdad = 1
falso = 0

# Primero hagamos las preguntas de la prueba
# Despues modificaremos esto para usar archivos en disco.
def obtener_preguntas():
    # observe como los datos quedan guardados en una lista de listas
    return [["Cual es el color de cielo en un dia claro?","azul"],\
            ["Cual es la respuesta a la vida, el universo y todas las cosas?","42"],\
            ["Cual es una palabra de cuatro letras para trampa de ratones?","gato"]]

# Esto hace una sola pregunta
# toma una sola pregunta
# devuelve verdad si el usuario escribio la respuesta correcta, si no, falso
def revisa_pregunta(pregunta_y_repuesta):
    #extrae la pregunta y la respuesta de la lista
    pregunta = pregunta_y_respuesta[0]
    respuesta = pregunta_y_respuesta[1]
    # da la pregunta al usuario
    repuesta_usuario = raw_input(pregunta)
    # compara la respuesta del usuario con la respuesta correcta
    if respuesa == respuesta_usuario:
        print "Correcto"
        return verdad
    else:
        print "Incorrecto, lo correcto era:",repuesta
        return falso

# Esto hara todas las preguntas
```

```
def examinar(preguntas):
    if len(preguntas) == 0:
        print "No hay preguntas."
        # el return sale de la funcion
        return
    indice = 0
    correctas = 0
    while indice < len(preguntas):
        #Revisa la pregunta
        if revisa_pregunta(preguntas[indice]):
            correctas = correctas + 1
        #ve a la siguiente pregunta
        indice = indice + 1
    #observe el orden del calculo, primero multiplica y luego divide
    print "Obtuvo ",correctas*100/len(preguntas),"% correcto de",len(questions)

#ahora hagamos el examen
examinar(obtener_preguntas())
```

Muestra:

```
Cual es el color de cielo en un dia claro?verde
Incorrecto, lo correcto era: azul
Cual es la respuesta a la vida, el universo y todas las cosas?42
Correcto
Cual es una palabra de cuatro letras para trampa de ratones?gato
Correcto
Obtuvo 66 % correcto de 3
```

8.4 Ejercicios

Expanda el programa prueba.py para que tenga un menú con la opción de tomar el examen, viendo la lista de preguntas y respuestas, y una opción para cerrar el programa. También, añada una nueva pregunta, "Que ruido hace una maquina verdaderamente avanzada?" con la respuesta "ping".

9. Bucle For

```
unoadiez = range(1,11)
for cuenta in unoadiez:
        print cuenta
```

y la siempre presente salida:

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

La salida es muy familiar, pero el código es muy diferente. El primer renglón usa la función `range`. La función `range` usa dos argumentos del siguiente modo: `range(comienzo,final)`. `comienzo` es el primer número producido. `final` es uno más que el último número. Observe que este programa puede ser más corto:

```
for cuenta in range(1,11):
        print cuenta
```

Estos son algunos ejemplos para mostrar lo que sucede con el comando range:

```
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(-32, -20)
[-32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21]
>>> range(5,21)
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> range(21,5)
[]
```

El siguiente renglón `for` cuenta `in` uno a diez: usa la estructura de control `for`. Una estructura de control `for` tiene la siguiente apariencia `for variable in lista:`. `lista` es recorrida empezando con el primer elemento hasta el final. A medida que `for` recorre cada elemento de una lista, pone cada uno dentro de variable. Esto permite que variable sea usada cada vez que el rizo `for` da una vuelta. He aquí otro ejemplo (este no tiene que escribirlo) como demostración:

```
demolista = ['vida',42, 'el universo', 6,'y',7,'todas las cosas']
for articulo in demolista:
    print "El articulo actualmente es:",
    print articulo
```

La salida es:

```
El articulo actualmente es: vida
El articulo actualmente es: 42
El articulo actualmente es: el universo
```

```
El articulo actualmente es: 6
El articulo actualmente es: y
El articulo actualmente es: 7
El articulo actualmente es: todas las cosas
```

Observe cómo el rizo `for` recorre la lista y sucesivamente hace que cada elemento sea el valor de articulo. (Vea cómo, si no desea que `print` empiece otro renglón, añada una coma al final del enunciado (p. ej., si desea imprimir algo en el mismo renglón).) Entonces, ¿para qué sirve `for`? (uf) El primer uso es para recorrer todos los elementos de una lista y hacer algo con ellos . Veamos un modo rápido para sumar todos los elementos:

```
lista = [2,4,6,8]
suma = 0
for num in lista:
        suma = suma + num
print "La suma es: ",suma
```

y la salida es simplemente:

```
La suma es:  20
```

O pudiera escribir un programa para descubrir si hay duplicados en una lista, tal como el siguiente programa:

```
lista = [4, 5, 7, 8, 9, 1,0,7,10]
lista.sort()
prev = lista[0]
del lista[0]
for cosa in lista:
        if prev == cosa:
                print "Encontre un duplicado de ",prev
        prev = cosa
```

y para que vean que sí funciona:

```
Encontre un duplicado de 7
```

Muy bien, entonces, ¿cómo funciona? Le presento una versión especial para depuración para explicar mejor (no necesita escribir esto):

```
l = [4, 5, 7, 8, 9, 1,0,7,10]
print "l = [4, 5, 7, 8, 9, 1,0,7,10]","\tl:",l
l.sort()
print "l.sort()","\tl:",l
prev = l[0]
print "prev = l[0]","\tprev:",prev
del l[0]
print "del l[0]","\tl:",l
for elemento in l:
        if prev == elemento:
                print "Encontre un duplicado de ",prev
        print "si prev == elemento:","\tprev:",prev,"\telemento:",elemento
        prev = elemento
        print "prev = elemento","\t\tprev:",prev,"\telemento:",elemento
```

y la salida es:

```
l = [4, 5, 7, 8, 9, 1,0,7,10]   l: [4, 5, 7, 8, 9, 1, 0, 7, 10]
#sort() reduce el numero de comparaciones,
#porque asi los duplicados seran elementos consecutivos
l.sort()        l: [0, 1, 4, 5, 7, 7, 8, 9, 10]
prev = l[0]     prev: 0
del l[0]        l: [1, 4, 5, 7, 7, 8, 9, 10]
si prev == elemento:        prev: 0         elemento: 1
prev = elemento             prev: 1         elemento: 1
si prev == elemento:        prev: 1         elemento: 4
prev = elemento             prev: 4         elemento: 4
si prev == elemento:        prev: 4         elemento: 5
prev = elemento             prev: 5         elemento: 5
si prev == elemento:        prev: 5         elemento: 7
prev = elemento             prev: 7         elemento: 7
Duplicate of  7  Found
si prev == elemento:        prev: 7         elemento: 7
prev = elemento             prev: 7         elemento: 7
si prev == elemento:        prev: 7         elemento: 8
prev = elemento             prev: 8         elemento: 8
si prev == elemento:        prev: 8         elemento: 9
prev = elemento             prev: 9         elemento: 9
si prev == elemento:        prev: 9         elemento: 10
prev = elemento             prev: 10        elemento: 10
```

La razón por la cual puse tantos enunciados `print` en el código es para que usted pueda ver lo que sucede a cada paso. (Por cierto, si alguna vez no puede descubrir porqué un programa no funciona, intente poniendo muchos enunciados `print` para que pueda ver lo que está sucediendo). Primero el programa empieza con una aburrida lista. Luego el programa ordena la lista, esto es para que los duplicados sean consecutivos. El programa luego declara la variable (para el valor) `prev(io)`. Luego, el primer elemento de la lista es borrado para no pensar incorrectamente que es un duplicado. Luego entramos en el rizo `for`. Cada elemento de la lista es comparado contra el valor previo. Si es igual, hemos encontrado un duplicado. Cambiamos entonces el valor de `prev` de modo que a la siguiente vuelta del rizo, el valor de `prev` sea el elemento previo al actual. Como era de esperarse, encontramos un duplicado para 7. (Vea cómo `\t` es usado para imprimir un "`tab`".)

El otro modo de usar el bucle `for` es para hacer algo un cierto número de veces. Aquí está el código para imprimir los primeros 11 números de la serie de Fibonacci:

```
a = 1
b = 1
for c in range(1,10):
        print a,
        n = a + b
        a = b
        b = n
```

y la sorprendente salida:

```
1 1 2 3 5 8 13 21 34
```

Todo lo que uno pueda hacer con los bucles `for` lo puede uno hacer con los bucles `while` pero los bucles `for` proporcionan un modo muy fácil de recorrer todos los elementos de una lista o hacer algo un cierto número de veces.

10. Expresiones Booleanas

El siguiente es un pequeño ejemplo de expresiones booleanas (pronuciado: "buleanas") (no tiene que escribirlo):

```
a = 6
b = 7
c = 42
print 1, a == 6
print 2, a == 7
print 3,a == 6 and b == 7
print 4,a == 7 and b == 7
print 5,not a == 7 and b == 7
print 6,a == 7 or b == 7
print 7,a == 7 or b == 6
print 8,not (a == 7 and b == 6)
print 9,not a == 7 and b == 6
```

Y la salida es:

```
1 1
2 0
3 1
4 0
5 1
6 1
7 0
8 1
9 0
```

¿Qué está pasando? El programa consiste en un montón de enunciados `print` muy raros. Cada enunciado `print` imprime un número y una expresión. El número es para identificar el enunciado `print` en cuestión. Observe cómo cada expresión resulta ser ya sea 0 ó 1. En Python, falso es escrito como 0 y verdadero como 1. Los renglones:

```
print 1, a == 6
print 2, a == 7
```

imprimen un 1 y un 0 respectivamente tal como era de esperar, ya que la primera comparación es verdadera y la segunda es falsa. El tercer `print`, `print 3,a == 6 and b == 7`, es un poco diferente. El operador `and` (en castellano: "y") significa que si ambas comparaciones (antes y después del `and`) son verdaderas entonces toda la expresión es verdadera, de otro modo toda la expresión es falsa. El siguiente renglón, `print 4,a == 7 and b == 7`, muestra cómo si parte de la una expresión `and` es falsa, toda la expresión es falsa. Podemos resumir el comportamiento de `and` del modo siguiente:

expresión	*resultado*
verdadero y verdadero	VERDADERO
verdadero y falso	FALSO
falso y verdadero	FALSO
falso y falso	FALSO

Note que si la primera expresión es falsa Python no evalúa la segunda ya que sabe que toda la expresión es falsa.

El siguiente renglón, `print 5,not a == 7 and b == 7`, usa el operador `not` (en castellano: "no"). `not` simplemente produce el opuesto de la expresión (Pudiéramos escribir la expresión como `print 5,a != 7 and b == 7`). Aquí está la tabla:

expresión	*resultado*
not verdadero	FALSO
not falso	VERDADERO

Los dos siguientes renglones, `print 6,a == 7 or b == 7` y `print 7,a == 7 or b == 6`, usan el operador `or` (en castellano: "o"). El operador `or` devuelve verdadero si la primera expresión es verdadera, o si la segunda expresión es verdadera o si ambas son verdaderas. Aquí está la tabla:

expresión	*resultado*
verdadero y verdadero	VERDADERO
verdadero y falso	VERDADERO
falso y verdadero	VERDADERO
falso y falso	FALSO

Vea que si la primera expresión es verdadera Python no evalúa la siguiente expresión porque sabe que toda la expresión es verdadera. Esto funciona porque or es verdadero si por lo menos una mitad de la expresión es verdadera. Si la primera parte es verdadera, la segunda parte puede ser verdadera o falsa y esto no afecta que toda la expresión sea verdadera.

Los siguientes dos renglones, `print 8,not (a == 7 and b == 6)` y `print 9,not a == 7 and b == 6`, muestran que podemos usar paréntesis para agrupar expresiones y forzar a que una parte sea evaluada primero. Observe que los paréntesis cambiaron la expresión de falsa a verdadera. Esto ocurró porque los paréntesis forzaron a que el `not` fuera aplicado a toda la expresión en lugar de sólo a la porción `a == 7`.

Aquí está un ejemplo del uso de una expresión booleana:

```
lista = ["Vida","El Universo","Todas las cosas","Juan","Juana","Vida","Juana"]

#haz una copia de la lista
copia = lista[:]
#ordena la copia
copia.sort()
prev = copia[0]
del copia[0]

cuenta = 0

#recorre la lista buscando una correspondencia
while cuenta < len(copia) and copia[cuenta] != prev:
    prev = copia[cuenta]
    cuenta = cuenta + 1

#Si no encontramos un elemento correspondiente, entonces
#cuenta no puede ser < len
#porque el rizo while continua mientras count es < len
#y no encontramos elementos correspondientes
if cuenta < len(copia):
    print "Primera correspondencia: ",prev
```

Y aquí está la salida:

```
Primera correspondencia: Juana
```

Este programa funciona revisando las correspondencias continuamente mientras la siguiente condición es verdadera `while count < len(copia) and copia[cuenta]`. Cuando ya sea que `cuenta` es mayor que el último índice de copia o ha encontrado un elemento correspondiente, el `and` deja de ser verdadero y el bucle termina. El `if` simplemente revisa que el `while` terminó porque encontró una correspondencia.

El otro 'truco' de `and` es usado en este ejemplo. Si ve la tabla de `and` notará que la tercera hilera es "falso y no revise". Si `cuenta >= len(copia)` (en otras palabras `cuenta < len(copia)` es falso) entonces `copia[cuenta]` no es evaluado. Esto es porque Python sabe que si la primera parte es falsa no pueden ser las dos verdaderas. Esto se conoce como un cortocircuito y es útil si la segunda parte del `and` causará un error si algo anda mal. Yo usé la primera expresión (`cuenta < len(copia)`) para revisar y ver si cuenta era un índice válido para copia. (Si no me cree, elimine los duplicados de 'Juana' y 'Vida', vea que todavía funciona y luego invierta el orden de `cuenta < len(copia) and copia[cuenta] != prev` a `copia[cuenta] != prev and cuenta < len(copia)`.)

Puede usar expresiones Booleanas cuando necesita revisar dos o más cosas diferentes al mismo tiempo.

10.2 Ejemplos

clave1.py

```
## Este programa pide al usuario un nombre y una clave de acceso.
# Luego los revisa para ver que el usuario puede entrar.

nombre = raw_input("Cual es su nombre? ")
clave = raw_input("Cual es la clave de acceso? ")
if nombre == "Juan" and clave == "Viernes":
    print "Bienvenido, Juan"
elif name == "Pedro" and clave == "Piedra":
    print "Bienvenido, Pedro"
else:
    print "No se quien es usted."
```

Muestras

```
Cual es su nombre? Juan
Cual es la clave de acceso? Viernes
Bienvenido, Juan

Cual es su nombre? Luis
Cual es la clave de acceso? Dinero
No se quien es usted.
```

10.3 Ejercicios

Escriba un programa que pida al usuario adivinar un nombre, pero que solo le dé tres oportunidades antes de que el programa termine.

11. Diccionarios

Este capítulo es acerca de *diccionarios*. Los diccionarios tienen **claves** y **valores**. Las claves se usan para encontrar los valores. Este es un ejemplo de un diccionario en acción:

```
def imprimir_menu():
    print '1. Imprimir numeros telefonicos'
    print '2. Agregar un numero telefonico'
    print '3. Borrar un numero telefonico'
    print '4. Buscar un numero telefonico'
    print '5. Cerrar el programa'
    print
numeros = {}
seleccion_menu = 0
imprimir_menu()
while seleccion_menu != 5:
    seleccion_menu = input("Escriba un numero (1-5):")
    if seleccion_menu == 1:
        print "Numeros Telefonicos:"
        for x in numeros.keys():
            print "Nombre: ",x," \tNumero: ",numeros[x]
        print
    elif seleccion_menu == 2:
        print "Agregar nombre y numero"
        nombre = raw_input("Nombre:")
        telefono = raw_input("Numero:")
        numeros[nombre] = telefono
    elif seleccion_menu == 3:
        print "Borrar nombre y numero"
        nombre = raw_input("Nombre:")
        if numeros.has_key(nombre):
            del numeros[nombre]
        else:
            print "No pude encontrar a ", nombre
    elif seleccion_menu == 4:
        print "Buscar numero"
        nombre = raw_input("Nombre:")
        if numeros.has_key(nombre):
            print "El numero es",numeros[nombre]
        else:
            print "No pude encontrar a ",nombre
    elif seleccion_menu != 5:
        imprimir_menu()
```

Y esta es la salida:

```
1. Imprimir numeros telefonicos
2. Agregar un numero telefonico
3. Borrar un numero telefonico
4. Buscar un numero telefonico
5. Cerrar el programa

Escriba un numero (1-5):2
Agregar nombre y numero
Nombre:Joe
Numero:545-4464
Escriba un numero (1-5):2
Agregar nombre y numero
```

```
Nombre:Jill
Numero:979-4654
Type in a number (1-5):2
Agregar nombre y numero
Nombre:Fred
Numero:132-9874
Escriba un numero (1-5):1
Numeros telefonicos:
Nombre:  Jill     Numero:  979-4654
Nombre:  Joe      Numero:  545-4464
Nombre:  Fred     Numero:  132-9874

Escriba un numero (1-5):4
Buscar numero
Nombre:Joe
El numero es 545-4464
Escriba un numero (1-5):3
Borrar nombre y numero
Nombre:Fred
Escriba un numero (1-5):1
Numeros telefonicos:
Nombre:  Jill     Numero:  979-4654
Nombre:  Joe      Numero:  545-4464

Escriba un numero (1-5):5
```

El programa es similar al anterior de la lista de nombres, que vimos en el capítulo de listas. El programa funciona del modo siguiente: Primero, definimos la función `imprimir_menu`. `imprimir_menu` solamente imprime un menu que luego usamos dos veces en el programa. Luego viene el renglón de apariencia extraña `numeros = {}`. Todo lo que ese renglón hace es decirle a Python que numeros es un diccionario. Los siguientes renglones hacen que el menú funcione. Los renglones:

```
for x in numeros.keys():
    print "Nombre: ",x," \tNumero: ",numeros[x]
```

recorren el diccionario e imprimen toda la información. La función `numeros.keys()` devuelve una lista empleada por el bucle `for`. La lista devuelta por `keys` no esta ordenada así que si desea orden alfabético debe ordenarla. De modo parecido a las listas, la declaración `numeros[x]` se usa para obtener un miembro específico del diccionario. Claro que en este caso `x` es una hilera. A continuación, el renglón `numeros[nombre] = telefono` agrega un nombre y un número de teléfono al diccionario. Si `nombre` ya hubiera existido en el diccionario, el contenido de `telefono` reemplazaría el valor anterior. Luego los renglones:

```
if numeros.has_key(nombre):
    del numeros[nombre]
```

ven si un nombre en particular está en el diccionario y lo borran si así era. La función `numeros.has_key(nombre)` devuelve verdadero si la clave `nombre` existe en el diccionario `numeros` o devuelve falso si no es así. El renglón `del numeros[nombre]` borra la clave nombre y el valor asociado con esa clave. Los renglones:

```
if numeros.has_key(nombre):
    print "El numero es",numeros[nombre]
```

revisa si el diccionario posee una clave en particular y, si dicha clave existe, entonces imprime el número telefónico asociado con dicha clave. Por último, si la selección del menú no es válida, imprime de nuevo el menú para que usted se divierta.

Revisión rápida: Los diccionarios tienen **claves** y **valores**. Las claves pueden ser hileras o números. Las claves apuntan hacia los valores. Los valores pueden se cualquier tipo de variable (incluyendo listas o hasta diccionarios (esos diccionarios o listas también pueden contener diccionarios o listas dentro de ellos (qué miedo ¿verdad? :))). Este es un ejemplo de un diccionario que contiene una lista:

```
puntos_max = [25,25,50,25,100]
asignaciones = ['tarea cap 1','tarea cap 2','examen   ','tarea cap 3','prueba']
estudiantes = {'#Max':puntos_max}

def imprimir_menu():
    print "1. Agregar un estudiante"
    print "2. Borrar un estudiante"
    print "3. Imprimir calificaciones"
    print "4. Registrar calificacion"
    print "5. Imprimir menu"
    print "6. Cerrar el programa"

def imprimir_todas_califs():
        print '\t',
        for i in range(len(asignaciones)):
            print asignaciones[i],'\t',
        print
        claves = estudiantes.keys()
        claves.sort()
        for x in claves:
            print x,'\t',
            calificaciones = estudiantes[x]
            imprimir_califs(calificaciones)

def imprimir_califs(calificaciones):
    for i in range(len(calificaciones)):
        print calificaciones[i],'\t\t',
    print

imprimir_menu()
seleccion_menu = 0
while seleccion_menu != 6:
    print
    seleccion_menu = input("Elija una opcion del menu (1-6):")
    if seleccion_menu == 1:
        nombre = raw_input("Agregar al estudiante:")
        estudiantes[nombre] = [0]*len(puntos_max)
    elif seleccion_menu == 2:
        nombre = raw_input("Borrar al estudiante:")
        if estudiantes.has_key(nombre):
            del estudiantes[nombre]
        else:
            print "El estudiante: ",nombre," no esta en el diccionario"
    elif seleccion_menu == 3:
        imprimir_todas_califs()
    elif seleccion_menu == 4:
        print "Registro de calificacion"
        nombre = raw_input("Estudiante:")
```

```
        if estudiantes.has_key(nombre):
            calificaciones = estudiantes[nombre]
            print "Escriba el la calificacion para registrar"
            print "Escriba 0 (cero) para salir"
            for i in range(len(asignaciones)):
                print i+1,' ',asignaciones[i],'\t',
            print
            imprimir_califs(calificaciones)
            cual = 1234
            while cual != -1:
                cual = input("Cual calificacion quiere cambiar:")
                cual = cual-1
                if 0 <= cual < len(calificaciones):
                    calificacion = input("Calificacion:")
                    calificaciones[cual] = calificacion
                elif cual != -1:
                    print "Calificacion no valida"
        else:
            print "No encontre a ese estudiante"
    elif seleccion_menu != 6:
        imprimir_menu()
```

y esta es una muestra de la salida:

```
1. Agregar un estudiante
2. Borrar un estudiante
3. Imprimir calificaciones
4. Registrar calificacion
5. Imprimir menu
6. Cerrar el programa

Elija una opcion del menu (1-6):3
       tarea cap 1    tarea cap 2    examen         tarea cap 3    prueba
#Max   25             25             50             25             100

Elija una opcion del menu:6

1. Agregar un estudiante
2. Borrar un estudiante
3. Imprimir calificaciones
4. Registrar calificacion
5. Imprimir menu
6. Cerrar el programa

Elija una opcion del menu:1
Agregar al estudiante:Bill

Elija una opcion del menu:4
Registro de calificacion
Estudiante:Bill
Escriba el la calificacion para registrar
Escriba 0 (cero) para salir
1   tarea cap 1       2   tarea cap 2       3   examen     4   tarea cap 3      5   prueba
0             0             0             0             0
Cual calificacion quiere cambiar:1
Calificacion:25
Cual calificacion quiere cambiar:2
Calificacion:24
Cual calificacion quiere cambiar:3
Calificacion:45
Cual calificacion quiere cambiar:5
Calificacion:95
Cual calificacion quiere cambiar:0

Elija una opcion del menu (1-6):3
       tarea cap 1    tarea cap 2    examen         tarea cap 3    prueba
#Max   25             25             50             25             100
Bill   25             24             45             0              95
```

```
Elija una opcion del menu (1-6):
```

El programa funciona de la siguiente manera. Básicamente, la variable estudiantes es un diccionario en el cual los nombres de los estudiantes sirven de claves y las calificaciones son los valores asociados a las claves. Los dos primeros renglones crean dos listas. El siguiente renglón `estudiantes = {'#Max':puntos_max}` crea un nuevo diccionario con la clave `#Max` y el valor asociado como `[25,25,50,25,100]` (dado que ese era el contenido de `puntos_max` cuando fue efectuada la declaración) (Uso la clave `#Max` porque el símbolo `#` tiene precedencia alfabética sobre todos los demás caracteres alfabéticos). Luego definimos `imprimir_menu`. A continuación definimos la función `imprimir_todas_califs` en los renglones:

```
def imprimir_todas_califs():
        print '\t',
        for i in range(len(asignaciones)):
            print asignaciones[i],'\t',
        print
        claves = estudiantes.keys()
        claves.sort()
        for x in claves:
            print x,'\t',
            calificaciones = estudiantes[x]
            imprimir_califs(calificaciones)
```

Observer cómo obtenemos primero las claves del diccionario estudiantes con la función `keys` en el renglón `claves = estudiantes.keys()`. `claves` es una lista, así que le podemos aplicar todas las funciones propias de las listas. Luego ordenamos las claves con la función `claves.sort()` ya que claves es una lista. `for` se usa para recorrer todas las claves. Las calificaciones quedan guardadas como una lista dentro de un diccionario, de modo que la asignación `calificaciones = estudiantes[x]` le da a la variable calificaciones la lista que está guardada en la clave `x`. La función `imprimir_califs` solamente imprime una lista y es definica unos cuantos renglones después.

Los renglones posteriores del programa instrumentan las varias opciones del menú. El renglón `estudiantes[nombre] = [0]*len(puntos_max)` agrega un estudiante a la clave de su nombre. La notación `[0]*len(puntos_max)` sólo crea un arreglo de ceros de la misma longitud que la lista `puntos_max`.

La opción para borrar un estudiante, borra un estudiante de modo similar al ejemplo del directorio telefonico. La opción para registrar calificaciones es un poco más complicada. En el renglón `calificaciones = estudiantes[nombre]` la variable calificaciones obtiene una referencia a las calificaciones del estudiante de nombre nombre. El registro de una calificación es efectuado en el renglón `calificaciones[cual] = calificacion`. Tal vez se haya fijado que el valor calificaciones nunca es asignado al diccionario de estudiantes (como sería el caso de `estudiantes[nombre] = calificaciones`). La razón por la que no hacemos eso es que calificaciones es de hecho otro nombre para `estudiantes[nombre]` y por eso, el simple hecho de cambiar calificaciones cambia `estudiante[nombre]`.

Los diccionarios proporcionan un modo fácil de asociar claves y valores. Esto puede usarse para seguir fácilmente el rastro de datos que están asociados a varias claves.

12. Usando Modulos

Este es el ejercicio de escritura de código de este capítulo (llámelo cal.py)[*]:

```
import calendar

anio = input("Escriba el numero del anio:")
calendar.prcal(anio)
```

Y esta es la salida que obtuve:

```
Escriba el numero del anio:2001
                                2001

      January                   February                  March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7                1  2  3  4                1  2  3  4
 8  9 10 11 12 13 14       5  6  7  8  9 10 11       5  6  7  8  9 10 11
15 16 17 18 19 20 21      12 13 14 15 16 17 18      12 13 14 15 16 17 18
22 23 24 25 26 27 28      19 20 21 22 23 24 25      19 20 21 22 23 24 25
29 30 31                  26 27 28                  26 27 28 29 30 31
```

(Eliminé parte de la salida del programa, pero esto es lo básico.) Entonces, ¿qué hace el programa? El primer renglón `import calendar` usa un comando nuevo llamado `import`. El comando `import` carga un módulo (en este caso el módulo de calendario, `calendar`). Para ver los comandos disponibles en los módulos regulares vea la biblioteca de referencia para Python (si la descargó junto con su instalación) o vaya a `http://www.python.org/doc/current/lib/lib.html` (en inglés). El módulo calendar está descrito en la sección 5.9 de la referencia. Si ve un poco más, la documentación lista una función llamada prcal que imprime el calendario para todo un año. El renglón calendar.prcal(year) usa esa función. En resumen, para usar un módulo hay que importarlo y luego usar la sintaxis nombre_del_módulo.función si queremos usar una función contenida en el módulo. Otro modo de escribir el programa es:

```
from calendar import prcal

anio = input("Escriba en numero de anio:")
prcal(anio)
```

Esta versión importa una función específica de un módulo. Aquí tiene otro programa que usa la Biblioteca de Python (llámelo algo así como reloj.py) (oprima Ctrl y la tecla 'c' simultáneamente para matar el programa):

[*] import de hecho busca un archivo llamado calendar.py y lo lee. Si usted llama calendar.py al archivo del programa que usted escribe 'import calendar' trataría de leerse a sí mismo, lo cual no funciona muy bien.
N. del T.: Si llama a su archivo 'calendario.py' el problema recién descrito no tiene por qué ocurrir. Por otro lado, es una pésima idea traducir los nombres de los módulos de la Biblioteca de Python a otro idioma, p. ej., calendar.py->calendario.py o time.py->tiempo.py ya que sus programas perderían la habilidad de Python para ser ejecutados en todas las plataformas que poseen un interpretador de Python.

```
from time import time, ctime

hora_anterior = ""
while(1):
    la_hora = ctime(time())
    if(hora_anterior != la_hora):
        print "La hora es:",ctime(time())
        hora_anterior = la_hora
```

Y algo de la salida es:

```
La hora es: Sun Aug 20 13:40:04 2000
La hora es: Sun Aug 20 13:40:05 2000
La hora es: Sun Aug 20 13:40:06 2000
La hora es: Sun Aug 20 13:40:07 2000
Traceback (innermost last):
  File "reloj.py", line 5, in ?
    the_time = ctime(time())
KeyboardInterrupt
```

El programa tiene un bucle sin salida así que tuve que cancelarlo oprimiendo Ctr+c. Dentro del bucle, el programa revisa si la hora ha cambiado e imprime un mensaje si la hora cambió. Observe cómo usamos varios nombres de funciones en el renglón donde usamos el módulo `import from time import time, ctime`.

La Biblioteca de Python contiene muchas funciones útiles. Estas funciones agregan habilidades a sus programas y muchas de ellas simplifican sobremanera programar en Python.

12.2 Ejercicios

Vuelva a escribir el programa *alto_bajo.py* de la sección 5.2 para usar los últimos dos dígitos de la hora (determinada en ese momento) en lugar de un número aleatorio.

13. Más acerca de Listas

Ya hemos visto listas y cómo podemos usarlas. Ahora que ya tiene más fundamentos puedo ahondar en las listas. Primero, veremos más modos de obtener los elementos en una lista y luego hablaremos acerca de cómo copiarlos.

He aquí algunos ejemplos del uso de índices para obtener un sólo elemento de una lista:

```
>>> lista = ['cero','uno','dos','tres','cuatro','cinco']
>>> lista[0]
'cero'
>>> lista[4]
'cuatro'
>>> lista[5]
'cinco'
```

Todos estos ejemplos debieran resultarle familiares. Si quiere el primer elemento en la lista vea al índice 0. El segundo elemento es índice 1 y así para toda la lista. Sin embargo, ¿qué tal si desea el último elemento de la lista? Un modo pudiera ser usar la función `len` como `lista[len(lista)-1]`. Este modo funciona porque la función `len` siempre devuelve el último índice más uno. El penúltimo elemento sería `lista[len(lista)-2]`. Hay un modo más fácil. En Python el último índice es siempre el índice -1. El penúltimo elemento es índice -2 y así sucesivamente. Vea algunos ejemplos:

```
>>> lista[len(lista)-1]
'cinco'
>>> lista[len(lista)-2]
'cuatro'
>>> lista[-1]
'cinco'
>>> lista[-2]
'cuatro'
>>> list[-6]
'cero'
```

Por tanto, cualquier elemento de la lista puede ser indexado de dos modos: desde el principio o desde el final.

Otro modo útil de obtener pedazos de listas es usando rebanadas. Veamos otro ejemplo para darle una idea de para qué son útiles:

```
>>> lista = [0,'Pedro',2,'S.P.A.M.','Media',42,"Juan","Juana"]
>>> lista[0]
0
>>> lista[7]
'Juana'
>>> lista[0:8]
[0,'Pedro',2,'S.P.A.M.','Media',42,"Juan","Juana"]
>>> lista[2:4]
[2, 'S.P.A.M.']
>>> lista[4:7]
['Media', 42, 'Juan']
>>> lista[1:5]
['Pedro', 2, 'S.P.A.M.', 'Media']
```

Las rebanadas sirven para devolver parte de una lista. El operador de rebanado tiene la forma `lista[primer_indice:siguiente_indice]`. La rebanada va desde el `primer_indice` al índice anterior al `siguiente_indice`. Puede usar ambos tipos de indexado:

```
>>> lista[-4:-2]
['Media', 42]
>>> list[-4]
'Media'
>>> list[-4:6]
['Media', 42]
```

Otro truco con rebanadas es el índice tácito. Si el primer índice no es especificado, Python supone que quiere desde el principio de la lista. Si el siguiente indice no es especificado, Python supone que quiere todo el resto de la lista. Algunos ejemplos:

```
>>> lista[:2]
[0, 'Pedro']
>>> lista[-2:]
['Juan', 'Juana']
>>> lista[:3]
[0, 'Pedro', 2]
>>> lista[:-5]
[0, 'Pedro', 2]
```

Vea un programa de ejemplo (copie y pegue la definición del poema, si así lo prefiere):

```
poema = ["<B>","Dona","Blanca","</B>","esta","cubierta","de","pilares","de","<B>",\
"oro","y","plata","</B>","romperemos","un","pilar", "para', "ver", "a", "<B>","Dona",\
"Blanca","</B>"]

def sacar_negritas(lista):
        verdadero = 1
        falso = 0
        ## es_negrita dice si estamos viendo a una sección en negrita del texto
        es_negrita = falso
        ## comienza_bloque es el indice del comienzo ya sea de un fragmento
        ## normal o un fragmento en negrita del texto
        comienza_bloque = 0
        for indice in range(len(lista)):
                ##Maneja comienzo de texto negrita
                if lista[indice] == "<B>":
                        if es_negrita:
                                print "Error:  Extra comienzo de negrita"
                        ##print "No negrita:",list[comienzo_bloque:indice]
                        es_negrita = verdadero
                        comienzo_bloque = indice+1
                ##Maneja final de texto negrita
                if lista[indice] == "</B>":
                        if not es_negrita:
                                print "Error: Extra cierre de negrita"
                        print "Negrita [",comienzo_bloque,":",indice,"] ",\
                        lista[comienzo_bloque:indice]
                        es_negrita = falso
                        comienzo_bloque = indice+1

sacar_negritas(poema)
```

y la salida es:

```
Negrita [ 1 : 3 ]  ['Dona', 'Blanca']
Negrita [ 10 : 13 ]  ['oro', 'y', 'plata']
Negrita [ 21 : 23 ]  ['Dona', 'Blanca']
```

La función `sacar_negritas` toma una lista donde cada elemento es una palabra o un símbolo. Los símbolos que busca con <B> el cual comienza texto en negrita y <\B> el cual termina texto en negrita. La función sacar_negritas recorre la lista y busca símbolos de comienzo y final.

El siguiente aspecto de las listas es copiarlas. Si trata algo sencillo como:

```
>>> a = [1,2,3]
>>> b = a
>>> print b
[1, 2, 3]
>>> b[1] = 10
>>> print b
[1, 10, 3]
>>> print a
[1, 10, 3]
```

Esto probablemente resulte desconcertante, ya que una modificación a bresultó en que también a cambiara. Lo que pasó es que la asignación `b = a` hace a `b` una referencia a `a`. Esto significa que podemos considerar a `b` como otro nombre para `a`. Por lo tanto, cualquier modificación a `b` también afecta a `a`. Sin embargo, algunas asignaciones no crean dos nombres para una lista:

```
>>> a = [1,2,3]
>>> b = a*2
>>> print a
[1, 2, 3]
>>> print b
[1, 2, 3, 1, 2, 3]
>>> a[1] = 10
>>> print a
[1, 10, 3]
>>> print b
[1, 2, 3, 1, 2, 3]
```

En este caso b no es una referencia a a porque la expresión `a*2` crea una lista nueva. Luego el enunciado `b = a*2` da a `b` una referencia a `a*2` más que una referencia a `a`. Todas las operaciones de asignación crean una referencia. Cuando usted pasa una lista como argumento de una función, también crea una referencia. La mayor parte del tiempo no tiene que preocuparse por crear referencias y no copias. Sin embargo, cuando necesite modificar una lista sin afectar a otra tiene que asegurarse que realmente haya creado una copia.

Hay varios modos para hacer una copia de una lista. El más simple, que funciona casi todo el tiempo es el operador rebanador ya que siempre hace una nueva lista aún cuando la rebanada abarca toda la lista:

```
>>> a = [1,2,3]
>>> b = a[:]
>>> b[1] = 10
>>> print a
[1, 2, 3]
>>> print b
[1, 10, 3]
```

Tomar la rebanada `[:]` crea una nueva copia de la lista. Sin embargo, solamente copia la lista más externa. Cualquier sublista contenida será una referencia a la sublista en la lista original. Por tanto, cuando la lista contiene otras listas como elementos, necesitamos copiar también las listas más internas. Puede hacerlo manualmente, pero Python contiene un módulo que lo hace. Use la función `deepcopy` del módulo `copy` :

```
>>> import copy
>>> a = [[1,2,3],[4,5,6]]
>>> b = a[:]
>>> c = copy.deepcopy(a)
>>> b[0][1] = 10
>>> c[1][1] = 12
>>> print a
[[1, 10, 3], [4, 5, 6]]
>>> print b
[[1, 10, 3], [4, 5, 6]]
>>> print c
[[1, 2, 3], [4, 12, 6]]
```

Primero que nada, note que a es un arreglo de arreglos (o lista de listas). Luego observe que cuando ejecutamos `b[0][1] = 10` tanto `a` como `b` cambian, pero `c` no. Esto es debido a que las listas más internas son referencias cuando hacemos rebanadas, pero con `deepcopy c` fue copiado totalmente.

Entonces, ¿debería preocuparme acerda de referencias cada vez que uso una función o una asignación? Las buenas noticias son que sólo tiene que preocuparse de las referencias cuando usa diccionarios o listas. Los números y las hileras crean referencias al momento de asignación, pero cada operación que modifica números e hileras crea una nueva copia de modo que no puede modificarlos inadvertidamente. Sí debe pensar acerca de referencias cuando modifique una lista o un diccionario.

En este momento quizá esté preguntándose porqué existen las referencias. La razón básica es velocidad. Es mucho más rápido hacer una referencia a una lista de mil elementos que copiar todos los elementos. La otra razón es que permite que una función modifique la lista o diccionario que recibe como parámetro. Sólo recuerde que las referencias existen si llega a tener algún problema extraño de datos que cambian cuando no debieran.

14. La Venganza de las Hileras

Y ahora, un truco bonito que podemos hacer con hileras:

```
def gritar(hilera):
    for letra in hilera:
        print "Denme un "+letra
        print "'"+letra+"'"

gritar("Pierde")

def enmedio(hilera):
    print "La letra de en medio es:",hilera[len(hilera)/2]

enmedio("abcdefg")
enmedio("El Lenguaje de Programacion Python")
enmedio("Atlanta")
```

Y la salida es:

```
Denme una P
'P'
Denme una i
'i'
Denme una e
'e'
Denme una r
'r'
Denme una d
'd'
Denme una e
'e'
La letra de enmedio es: d
La letra de enmedio es: o
La letra de enmedio es: a
```

Lo que estos programas demuestran es que las hileras son similares a las listas de varios modos. El procedimiento `gritar` muestra que es posible usar los bucles `for` con hileras tal como con listas. El procedimiento enmedio muestra que con hileras también podemos usar la función `len` así como índices y rebanadas. La mayoría de las propiedades de las listas también funcionan con hileras.

El siguiente programa demuestra aspectos específicos de las hileras:

```
def a_mayuscula(hilera):
    ## Convierte la hilera a letras mayusculas
    mayuscula = ""
    for letra in hilera:
        if 'a' <= letra <= 'z':
            ubicacion = ord(letra) - ord('a')
            nuevo_ascii = ubicacion + ord('A')
            letra = chr(nuevo_ascii)
        mayuscula = mayuscula + letra
    return mayuscula

print a_mayuscula("Este es texto")
```

con la salida:

```
ESTE ES TEXTO
```

Esto funciona porque la computadora representa las letras de una hilera como números de 0 a 255. Python tiene una función llamada `ord` (apócope de ordinal) que devuelve el número que representa una letra. También hay una función correspondiente llamada `chr` que devuelve la letra que corresponde a un número. Con esto en mente, el funcionamiento del programa debería empezar a aclararse. El primer detalle es el renglón : `if 'a' <= letra <= 'z':` el cual revisa si la letra es minúscula. Si lo es entonces usa los siguientes renglones. Primero la letra es convertida a una ubicación, de modo que `a=0,b=1,c=2` y así sucesivamente, con el renglón: `ubicacion = ord(letra) - ord('a')`. Luego encuentra el nuevo valor (el número que representa la letra mayúscula) con `nuevo_ascii = ubicacion + ord('A')`. Este valor luego es convertido a una letra, que ahora ya es mayúscula.

Ahora hagamos un ejercicio interactivo escrito:

```
>>> #Entero a Hilera
...
>>> 2
2
>>> repr(2)
'2'
>>> -123
-123
>>> repr(-123)
'-123'
>>> #Hilera a Intero
...
>>> "23"
'23'
>>> int("23")
23
>>> "23"*2
'2323'
>>> int("23")*2
46
>>> #Punto flotante a Hilera
...
>>> 1.23
1.23
>>> repr(1.23)
'1.23'
>>> #Punto flotante a Entero
...
>>> 1.23
1.23
>>> int(1.23)
1
>>> int(-1.23)
-1
>>> #Hilera a Punto flotante
...
>>> float("1.23")
1.23
>>> "1.23"
```

```
'1.23'
>>> float("123")
123.0
```

Por si no lo ha adivinado, la función `repr` puede convertir un entero a una hilera y la función `int` puede convertir una hilera a un entero. La función `float` puede convertir una hilera a un número de punto flotante. La función `repr` devuelve una representación, que podemos imprimir, de algo. Veamos algunos ejemplos:

```
>>> repr(1)
'1'
>>> repr(234.14)
'234.14'
>>> repr([4,42,10])
'[4, 42, 10]'
```

La función `int` trata de convertir una hilera (o un punto flotante) a un entero. Hay una función similar llamada `float` que convertirá un entero o una hilera a un punto flotante. Otra función en Python es `eval`. La función `eval` toma una hilera y devuelve datos del tipo de Python cree que encontró. Por ejemplo:

```
>>> v=eval('123')
>>> print v,type(v)
123 <type 'int'>
>>> v=eval('645.123')
>>> print v,type(v)
645.123 <type 'float'>
>>> v=eval('[1,2,3]')
>>> print v,type(v)
[1, 2, 3] <type 'list'>
```

Si usa la función `eval` debiera revisar que devuelve el tipo que usted espera.

Una función de hileras muy útil es la función `split`. Un ejemplo:

```
>>> import string
>>> string.split("Este es un monton de palabras")
['Este', 'es', 'un', 'monton', 'de', 'palabras']
>>> string.split("Primer lote, segundo lote, tercero, cuarto",",")
['Primer lote', ' segundo lote', ' tercero', ' cuarto']
```

Observe cómo `split` convierte una hilera en una lista de hileras. La hilera es dividida por espacios por definición o por segundo argumento, el cual es opcional (una coma, en este caso).

14.2 Ejemplos

```
#Este programa requiere una excelente comprension de los numeros decimales
def a_hilera(entero_entrada):
    "Convierte un entero a una hilera"
    hilera_salida = ""
    prefijo = ""
    if entero_entrada < 0:
        prefijo = "-"
        entero_entrada = -entero_entrada
    while entero_entrada / 10 != 0:
        hilera_salida = chr(ord('0')+entero_entrada % 10) + hilera_salida
        entero_entrada = entero_entrada / 10
    out_str = chr(ord('0')+entero_entrada % 10) + hilera_salida
    return prefix + hilera_salida

def a_entero(hilera_entrada):
    "Convierte una hilera a un entero"
    numero_salida = 0
    if hilera_entrada[0] == "-":
        factor = -1
        hilera_entrada = hilera_entrada[1:]
    else:
        factor = 1
    for x in range(0,len(hilera_entrada)):
        numero_salida = numero_salida * 10 + ord(in_str[x]) - ord('0')
    return numero_salida * factor

print a_hilera(2)
print a_hilera(23445)
print a_hilera(-23445)
print a_entero("14234")
print a_entero("12345")
print a_entero("-3512")
```

La salida es:

```
2
23445
-23445
14234
12345
-3512
```

15. Lectura y escritura de archivos

Aquí tiene un ejemplo sencillo de lectura y escritura de archivos:

```
#Escribe un archivo
archivo_salida = open("prueba.txt","w")
archivo_salida.write("Este texto va para el archivo de salida\nVea el archivo y
compruebelo\n")
archivo_salida.close()

#Lee un archivo
archivo_entrada = open("prueba.txt","r")
texto = archivo_entrada.read()
archivo_entrada.close()

print texto,
```

La salida y el contenido del archivo prueba.txt son:

```
Este texto va para el archivo de salida
Vea el archivo y compruebel
```

Observe que escribió un archivo llamado prueba.txt en el directorio desde el cual ejecutó el programa. El `\n` en la hilera le dice a Python que introduzca un retorno de carro en su lugar (nuevo renglón).

Una vista rápida de lectura y escritura de archivos es:

1. Cree un objeto de archivo con la función open.
2. Lea o escriba al objeto de archivo (dependiendo de cómo lo abrió)
3. Ciérrelo

El primer paso es obtener un objeto de archivo. Para hacer esto usamos la función `open`. El formato es `objeto_archivo = open(nombrearchivo,modo)` donde `objeto_archivo` es la variable para poner el objeto archivo, nombrearchivo es una hilera con el nombre del archivo, y modo es ya sea "r" para leer un archivo o "w" para escribir en un archivo. Luego podemos llamar las funciones de los objetos de archivo. Las dos más comunes son `read` y `write`. La función `write` añade una hilera al final del archivo. La función `read` lee la siguiente cosa en el archivo y la devuelve como una hilera. Si no proporcionamos argumentos, devolverá todo el contenido del archivo (tal como en el ejemplo).

Ahora veamos una nueva versión del programa de números telefónicos que hicimos anteriormente:

```
import string

verdadero = 1
falso = 0

def imprimir_numeros(numeros):
    print "Numeros telefonicos:"
    for x in numeros.keys():
        print "Nombre: ",x," \tNumero: ",numeros[x]
    print
```

```
def agregar_numero(numeros,nombre,numero):
    numeros[nombre] = numero

def buscar_numero(numeros,nombre):
    if numeros.has_key(nombre):
        return "El numero es "+numeros[nombre]
    else:
        return "No pude hallar a "+nombre

def eliminar_numero(numeros,nombre):
    if numeros.has_key(nombre):
        del numeros[nombre]
    else:
        print "No pude hallar a ", nombre


def cargar_numeros(numeros,nombrearchivo):
    archivo_entrada = open(nombrearchivo,"r")
    while verdadero:
        renglon_entrada = archivo_entrada.readline()
        if renglon_entrada == "":
            break
        renglon_entrada = renglon_entrada[:-1]
        [nombre,numero] = string.split(renglon_entrada,",")
        numeros[nombre] = numero
    archivo_entrada.close()

def guardar_numeros(numeros,nombrearchivo):
    archivo_salida = open(nombrearchivo,"w")
    for x in numeros.keys():
        archivo_salida.write(x+","+numeros[x]+"\n")
    archivo_salida.close()


def imprimir_menu():
    print '1. Imprimir numeros telefonicos'
    print '2. Agregar numero telefonico'
    print '3. Borrar numero telefonico'
    print '4. Buscar numero telefonico'
    print '5. Cargar numeros'
    print '6. Guardar numeros'
    print '7. Cerrar el programa'
    print

lista_telefonos = {}
seleccion_menu = 0
imprimir_menu()
while seleccion_menu != 7:
    seleccion_menu = input("Escriba su seleccion (1-7):")
    if seleccion_menu == 1:
        imprimir_numeros(lista_telefonos)
    elif seleccion_menu == 2:
        print "Agregue nombre y numero"
        nombre = raw_input("Nombre:")
        telefono = raw_input("Numero:")
        agregar_numero(lista_telefonos,nombre,telefono)
    elif seleccion_menu == 3:
        print "Borrar nombre y numero"
        nombre = raw_input("Nombre:")
```

```
        borrar_numero(lista_telefonos,nombre)
    elif seleccion_menu == 4:
        print "Buscar numero"
      nombre = raw_input("Nombre:")
        print buscar_numero(lista_telefonos,nombre)
    elif seleccion_menu == 5:
        nombrearchivo = raw_input("Archivo para cargar:")
        cargar_numeros(lista_telefonos,nombrearchivo)
    elif seleccion_menu == 6:
        nombrearchivo = raw_input("Archivo en el cual guardar:")
        guardar_numeros(lista_telefonos,nombrearchivo)
    elif seleccion_menu == 7:
        pass
    else:
        imprimir_menu()
print "Adios"
```

Observe que ahora incluye guardar y cargar archivos. Aquí tiene la salida de las dos veces que lo ejecuté:

```
> python telefonos2.py
1. Imprimir numeros telefonicos
2. Agregar numero telefonico
3. Borrar numero telefonico
4. Buscar numero telefonico
5. Cargar numeros
6. Guardar numeros
7. Cerrar el programa

Escriba su seleccion (1-7):2
Agregue nombre y numero
Nombre:Juana
Numero:1234
Escriba su seleccion (1-7):2
Agregue nombre y numero
Nombre:Pedro
Numero:4321
Escriba su seleccion (1-7):1
Numeros telefonicos:
Nombre:  Pedro    Numero:  4321
Nombre:  Juana    Numero:  1234

Escriba su seleccion (1-7):6
Archivo en el cual guardar:numeros.txt
Escriba su seleccion (1-7):7
Adios

> python telefonos2.py
1. Imprimir numeros telefonicos
2. Agregar numero telefonico
3. Borrar numero telefonico
4. Buscar numero telefonico
5. Cargar numeros
6. Guardar numeros
7. Cerrar el programa

Escriba su seleccion (1-7):5
Archivo para cargar:numeros.txt
Escriba su seleccion (1-7):1
```

```
Numeros telefonicos:
Nombre:  Pedro    Numero:  4321
Nombre:  Juana    Numero:  1234

Escriba su seleccion (1-7):7
Adios
```

Las nuevas porciones de este programa son:

```
def cargar_numeros(numeros,nombrearchivo):
    archivo_entrada = open(nombrearchivo,"r")
    while verdadero:
        renglon_entrada = archivo_entrada.readline()
        if renglon_entrada == "":
            break
        renglon_entrada = renglon_entrada[:-1]
        [nombre,numero] = string.split(renglon_entrada,",")
        numeros[nombre] = numero
    archivo_entrada.close()

def guardar_numeros(numeros,nombrearchivo):
    archivo_salida = open(nombrearchivo,"w")
    for x in numeros.keys():
        archivo_salida.write(x+","+numeros[x]+"\n")
    archivo_salida.close()
```

Veamos la porción para guardar números. Primero crea un objeto archivo con la función `open(nombrearchivo,"w")`. Luego lo recorre y crea un renglón para cada número telefónico con la función `archivo_salida.write(x+","+numeros[x]+"\n")`. Esto escribe un renglón que contiene el nombre, una coma, el número y empieza un renglón nuevo.

La porción de carga es un poco más complicada. Empieza obteniendo el objeto archivo. Entonces usa un bucle `while 1:` para mantenerse dentro del bucle hasta que encuentre un enunciado `break`. Luego lee un renglón al ejecutar `renglon_entrada = archivo_entrada.readline()`. La función `getline` devolverá un una hilera vacía `(len(string) == 0)` cuando llegue al final del archivo. El enunciado `if` revisa si esto ha ocurrido y usa el `break` para salir del bucle cuando ha llegado al final del archivo. Por supuesto que si la función `readline` no devolviera el retorno de carro al final del renglón, no habría modo de saber si una hilera vacía era un renglón vacío o el final del archivo así que el retorno de carro es conservado en lo que devuelve `getline`. Por lo tanto, tenemos que deshacernos del retorno de carro. El renglón en `renglon_entrada = renglon_entrada[:-1]` hace esto eliminando la última letra. A continuación, el `renglón[nombre,numero] = string.split(renglon_entrada,",")` divide el renglón usando la coma como separador en un nombre y un número. Este par es luego añadido al diccionario numeros.

15.2 Ejercicios

Ahora modifique el programa de calificaciones de la sección 11 para que escriba y lea archivos que contengan los registros de los estudiantes.

16. Manejo de la imperfección (o qué hacer con los errores)

Así que ya tiene el programa perfecto, funciona sin errores, excepto por un pequeño detalle: se muere si el usuario introduce información no válida. No tema, porque Python tiene una estructura de control especial para usted. Se llama `try` (intentar) e intenta hacer "algo". Aquí tiene un ejemplo de un programa con un problema:

```
print "Escriba Control C or -1 para cerrar el programa"
numero = 1
while numero != -1:
    numero = int(raw_input("Escriba un numero: "))
    print "Usted escribio: ",numero
```

Observer ahora, cuando usted escribe @#& la salida es algo así:

```
Traceback (innermost last):
  File "try_less.py", line 4, in ?
    number = int(raw_input("Enter a number: "))
ValueError: invalid literal for int(): @#&
```

Como puede ver, la función `int` no está nada contenta con el número @#& (tal como debe ser). El último renglón muestra cuál es el problema; Python encontró un `ValueError` (Valor Erróneo). ¿Cómo puede manejar esto en su programa? Lo primero que hacemos es: ponemos el lugar donde ocurren los errores en un bloque `try`, y luego: le decimos a Python qué queremos que haga con los errores de tipo `ValueError`. El siguiente programa hace esto:

```
print "Escriba Control C or -1 para cerrar el programa"
numero = 1
while numero != -1:
    try:
        numero = int(raw_input("Escriba un numero: "))
        print "Usted escribio: ",number
    except ValueError:
        print "Eso no es un numero."
```

Ahora, cuando ejecutamos el nuevo programa y le damos un @#& nos dice ``Eso no es un número." y continúa con lo que estaba haciendo.

Cuando su programa tiene algún error que usted sabe cómo manejar, pónga el código en un bloque `try`, y ponga el modo de manejar el error en el bloque `except`.

16.2 Ejercicios

Actualice por lo menos el programa de números telefónicos para que no truene si el usuario escribe cualquier dato en el menú.

17. Fin

Al momento presente estoy trabajando en añadir más secciones a este documento. Por el momento le recomiendo ojear *The Python Tutorial* por Guido van Rossum. Si no tiene problemas con el inglés, debería ser capaz de entender una muy buena parte.

Este tutor es una obra inacabada todavía. Quiero cualquier comentario que pueda tener acerca de aspectos que le gustaría ver descritos. Los mensajes de agradecimiento son notorios por ayudarme a: terminar cantidades masivas de trabajo, agregar mejoras y escuchar cuidadosamente a comentarios. :).

Feliz programación, que le sirva para cambiar su vida y el mundo.

Por `hacer=[ 'errores','cómo hacer módulos','más acerca de rizos','más acerca de hileras', 'lectura y escritura de archivos','cómo usar ayuda en línea','try','pickle','cualquier cosa que alguien piense que es buena idea']`

18. Preguntas frecuentes

1. No puedo usar programas que soliciten entrada de datos (input).
   Si está usando IDLE mejor intente con la línea de comandos. Este problema parece habers sido solucionado in IDLE 0.6 y versiones más recientes. Si está usando una versión más antigua de IDLE intente actualizarse a Python 2.0 o más reciente.
2. No puedo escribir programas de más de un renglón.
   Si el programa empieza a ejecutar tan pronto como escribe algo, necesita escribirlo en un archivo en lugar de usar el modo interactivo. (Pista: el modo interactivo es el modo con el prompt >>>.)
3. Mi pregunta no está contestada aquí.
   Envíeme un mensaje y pregúnteme. Por favor envíeme el código fuente si es relevante (aún, (o quizás especialmente) si no funciona). Información útil para incluir son: lo que estaba tratando de hacer, lo que sucedió, lo que usted esperaba que sucediera, mensajes de error, versión de Python, Sistema Operativo, y si su gato estaba pisando el teclado. (El gato de mi casa siente un especial cariño por barras espaciadoras y teclas de control.)
4. ¿Cómo hago una interfase gráfica para el usuario en Python?
   Puede usar Tkinter en `http://www.python.org/topics/tkinter/` o WXPython en `http://www.wxpython.org/`
5. ¿Cómo hago un juego en Python?
   Probablemente el mejor método sea usar PYgame en `http://pygame.org/`
6. ¿Cómo hago un ejecutable de un programa de Python?
   Respuesta corta: Python es un lenguaje interpretado así que es imposible hacer un ejecutable. Respuesta larga: es posible crear algo similar a un ejecutable tomando el interpretador de Python y el programa y distribuyéndolos unidos. Para saber más acerca de este problema, vea (en inglés): `http://www.python.org/cgi-bin/faqw.py?req=all#4.28`
7. Necesito ayuda con los ejercicios
   Pista, el programa de la clave de acceso requiere dos variables, una para llevar la cuenta del número de veces que la clave ha sido escrita y la otra para guardar la última clave escrita. También puede descargar soluciones de `http://www.honors.montana.edu/~jjc/easytut/`

Acerca de este documento ...

Este documento fue traducido al castellano por Víctor M. Rosas-García. Lo que sigue se refiere a la versión en inglés;

Non-Programmers Tutorial For Python

This document was generated using theLaTeX2HTML translator Version 2K.1beta (1.48)

The command line arguments were:
latex2html -split 3 -local_icons -address 'Josh Cogliati jjc@honors.montana.edu' easytut.tex

The translation was initiated by Josh Cogliati on 2002-09-08