Introducción a la programación de computadores

mediante el lenguaje Python

Parte II. Aspectos Complementarios

Á. Ibeas

Buena parte de estas notas están inspiradas en los materiales siguientes:

- J. Campbell, P. Gries, J. Montojo y G. Wilson: **«Practical Programming»**. Pragmatic Bookshelf, 2009.
- `http://docs.python.org`
- Andrés Marzal e Isabel Gracia: **«Introducción a la programación con Python»**. Universitat Jaume I.

1. Objetos, métodos y atributos
2. Manejo de archivos
3. Bases de datos
4. Programación orientada a objetos
5. Interfaces gráficas

Objetos y métodos

Hemos encontrado alguna instrucciones que se comportaba como una función pero cuya invocación se hacía de manera algo diferente:

```
>>> texto="torrelavega"
>>> texto.capitalize()
'Torrelavega'
```

Un **método** es una función que se asocia a un determinado tipo de datos (**objeto**). Es necesario llamar a los métodos desde un objeto concreto, que entrará a formar parte de los argumentos.

```
>>> a=complex(3,-2)
>>> a.conjugate()
(3+2j)
```

Del mismo modo que los métodos son funciones «propias» de un objeto, algunos objetos se componen de varios datos, que reciben el nombre de «atributos». Podemos acceder a los atributos de un objeto de manera similar a como llamamos a sus métodos, eliminando los paréntesis, ya que los atributos no son funciones. En general, no se pueden modificar directamente los atributos de un objeto:

```
>>> a=complex(3,2)
>>> a.real
3.0
>>> a.real=5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: readonly attribute
>>> a=complex(5,2);a
(5+2j)
```

1. Objetos, métodos y atributos
2. Manejo de archivos
3. Bases de datos
4. Programación orientada a objetos
5. Interfaces gráficas

Archivos

Todos estamos familiarizados con el sistema de archivos que los ordenadores utilizan para almacenar información. Es útil conocer cómo podemos acceder desde un programa a los datos que contiene un archivo para manejarlos de manera automatizada.

También es conveniente almacenar en ficheros los datos que devuelve un algoritmo, para poder utilizalos con posterioridad, publicarlos o compartirlos con otras aplicaciones.

Analizaremos algunas operaciones básicas de edición de archivos de texto, tratándolos como objetos dentro de un programa Python.

Supongamos que disponemos del fichero siguiente:

gastos.txt

```
1500
350
2600
```

Veamos, con el siguiente fragmento de código, como acceder a su contenido:

```
fichero=open("gastos.txt","r")
total=0
for linea in fichero:
    num=int(linea)
    total+=num
fichero.close()
print "TOTAL",total
```

Destaquemos tres instrucciones del programa anterior:

- `fichero=open("gastos.txt","r")`
  Con esta instrucción «preparamos» el archivo para ser leído (parámetro «r» —read—) y lo asociamos a la variable `fichero`.
- `fichero.close()`
  Es importante «cerrar» los archivos que ya no vamos a utilizar.
- `for linea in fichero:`
  Un fichero de lectura puede interpretarse como una lista formada por cadenas de texto (sus líneas).

Existen dos modos adicionales a «r» (lectura) para abrir un archivo: «w» (escritura) y «a», para añadir datos al final de un archivo. Con el modo «w» se borran los datos que el archivo contuviera anteriormente.

```
print("Vayan escribiendo sus nombres.")
print('Escriban "FIN" cuando no quede nadie más.')
cadena="PERSONAL\n"
fichero=open("lista.txt","w")
while cadena!="FIN":
    fichero.write(cadena+"\n")
    cadena=raw_input()
```

1 Objetos, métodos y atributos

2 Manejo de archivos

3 Bases de datos

4 Programación orientada a objetos

5 Interfaces gráficas

Una base de datos proporciona un sistema de almacenamiento de información que permite acceder a los datos y operar con ellos en general de manera más eficaz que un simple fichero de texto.

Utilizaremos el módulo `sqlite3` de Python, que proporciona funciones para crear y manejar bases de datos con el sistema SQLite.

```
import sqlite3
```

Del mismo modo que un archivo se representa mediante un objeto (del tipo `file`), usaremos objetos del tipo `Connection` para las bases de datos:

```
b=sqlite3.connect("datos")
```

Un último movimiento antes de empezar: necesitaremos un objeto del tipo `cursor`.

```
c=b.cursor()
```

Ya estamos listos para hacer operaciones. Si el archivo `datos` almacenaba una base de datos, podemos consultarla. En cualquier caso, podemos introducir datos. . .

Ejecución de comandos

Las bases de datos SQLite se manejar mediante comandos del lenguaje SQL. Ilustraremos mediante ejemplos algunos comandos sencillos:

```
c.execute('CREATE TABLE Godos(No TEXT, Co INTEGER, Fin INTEGER)')
```

Con esto hemos creado una «tabla» con tres columnas o **campos**: una contendrá texto y las otras dos, números. Utilizamos los tipos de datos de SQLite (`TEXT`, `INTEGER`,...), que no los de Python. Una vez creada la tabla, pasemos a rellenarla:

```
c.execute('INSERT INTO Godos VALUES("Leovigildo",571,586)')
c.execute('INSERT INTO Godos VALUES("Recaredo",586,601)')
c.execute('INSERT INTO Godos VALUES("Chindasvinto",642,653)')
c.execute('INSERT INTO Godos VALUES("Recesvinto",649,672)')
c.execute('INSERT INTO Godos VALUES("Wamba",672,680)')
```

Hasta el momento, a pesar de haber ido insertando «registros» en la base de datos, no hemos modificado el fichero datos al que la habíamos asociado. Tampoco podemos acceder a esos datos. Es necesario ejecutar antes la instrucción:

```
b.commit()
```

Vamos a ordenar estos reyes por orden alfabético:

```
>>>c.execute('SELECT No FROM Godos ORDER BY No')
<sqlite3.Cursor object at 0x7fa8bba70c38>
>>> c.fetchall()
[(u'Chindasvinto',), (u'Leovigildo',), (u'Recaredo',), (u'Re
```

Busquemos ahora los reyes cuyo reinado comenzó en la segunda mitad del siglo VII:

```
>>>c.execute('SELECT No, Fin FROM Godos WHERE Co > 650')
<sqlite3.Cursor object at 0x7fa8bba70c38>
>>>c.fetchall()
[(u'Wamba', 680)]
```

Teniendo en cuenta que el módulo que estamos manejando devuelve objetos de Python, modemos utilizar estos comandos dentro de programas complejos, aprovechar su salida para realizar otros cálculos. . .

```
c.execute('SELECT No, Co, Fin FROM Godos ORDER BY Co')
lista=c.fetchall()
contador=1
for i in lista:
    cad="epitafio_"+str(contador)+".txt"
    fich=open(cad,"w")
    fich.write("El excelso "+str(i[0])+" ciñó \
la corona durante "+str(i[2]-i[1])+" gloriosos \
años.\n")
    fich.close()
    contador+=1
```

1 Objetos, métodos y atributos

2 Manejo de archivos

3 Bases de datos

4 Programación orientada a objetos

5 Interfaces gráficas

Hasta ahora hemos manejado objetos de varios tipos, tipos proporcionados por el lenguaje Python que admiten métodos propios. El programador también tiene la posibilidad de definir sus propios tipos de datos, mediante una **clase**. Esto resulta de utilidad cuando se quiere diseñar un programa que hace uso habitual de ese tipo de datos (pensemos en polinomios, grafos, pedidos de un negocio ...) mediante operaciones específicas (suma de polinomios, dibujo de grafos, gestión de pedidos...)

Veamos, por ejemplo, cómo definir una clase que permita manejar vectores en dos dimensiones:

```
class vector(object):
    '''Vectores bidimensionales'''
```

Obviamente, la declaración del código anterior resulta de poca utilidad. A pesar de esto, podemos crear objetos «miembros» de los clase y dotarles de atributos:

```
>>> v=vector()
>>> v.x=3
>>> v.y=-5
>>> v.autor="Marta"
>>> v
<__main__.vector object at 0x8384fcc>
>>> math.sqrt(v.x**2+v.y**2)
5.8309518948453007
```

Declaración de métodos

Vamos ahora a definir algunos métodos para hacer operaciones con los objetos de la clase:

```
import math
class vector(object):
    '''Vectores bidimensionales'''
    def norma(self):
        return math.sqrt(self.x**2+self.y**2)
```

El método `norma` tiene un único argumento: el objeto desde el que se invoca:

```
>>> v.x=1; v.y=-1
>>> v.norma()
1.4142135623730951
```

Pero podemos definir métodos con más argumentos, siempre «dentro» de la declaración de la clase correspondiente:

```
class vector(object):
    (...)
    def prod_escalar(self,otro):
        return self.x*otro.x+self.y*otro.y
```

Al invocar un método, el primer argumento es el objeto desde el que se llama y el resto se colocan entre paréntesis:

```
>>>w=vector(); w.x=2; w.y=-1
>>>v.prod_escalar(w)
3
```

Hay algunos métodos especiales que se invocan de manera distinta. El más útil es `__init__`, que se ejecuta al crear un objeto. Pongamos:

```
class vector(object):
    '''Vectores bidimensionales'''
    def __init__(self,abs,ord):
        self.x=abs
        self.y=ord
```

Con este método, podemos crear objetos del modo siguiente:

```
>>>v=vector(1,-1)
```

Evitamos así la necesidad de asignar los atributos y garantizamos que los objetos creados tienen los atributos necesarios.

Veamos ahora cómo controlar el comportamiento de nuestros objetos como argumentos de la instrucción `print`:

```
def __str__(self):
    return "(%f,%f)" % (self.x,self.y)
```

```
>>>print vector(5,-6)
>>>(5.000000,-6.000000)
```

También podemos determinar a nuestro gusto el comportamiento de los operadores +, <,. . . :

```
def __add__(self,otro):
    return vector(self.x+otro.x,self.y+otro.y)
def __lt__(self,otro):
    return self.norma()<otro.norma()
```

1. Objetos, métodos y atributos
2. Manejo de archivos
3. Bases de datos
4. Programación orientada a objetos
5. Interfaces gráficas

En esta sección ejemplificamos cómo, con un poco de esfuerzo adicional, se puede sustituir la interfaz de comunicación entre el usuario y el programa que veníamos utilizando hasta ahora (basada en la línea de comandos y las instrucciones `print` e `input`) por una interfaz gráfica, basada en ventanas, que puede parecer más cómoda para el usuario.

En primer lugar, cargamos el módulo que utilizaremos y creamos una ventana vacía:

programa.py

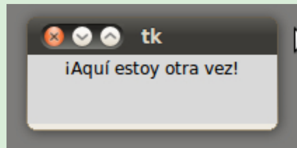
```
#!/usr/bin/python

from Tkinter import *
ventana=Tk()
ventana.mainloop()
```

El código siguiente coloca un fragmento de texto en la ventana. En general, se trata de añadir «widgets»: textos, recuadros para que el usuario introduzca datos, botones...

programa.py

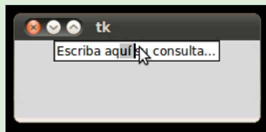
```
from Tkinter import *
ventana=Tk()
chisme=Label(ventana,text="¡Aquí estoy otra vez!")
chisme.pack()
ventana.mainloop()
```



El «widget» `Entry` permite al usuario introducir texto. Al contrario que en el ejemplo anterior, no lo asociamos con un objeto de tipo `str`, cuyo valor es inmutable, sino con uno del tipo `StringVar`, proporcionado por el módulo `Tkinter`, cuyo contenido se puede leer y escribir mediante los métodos `get` y `set`, respectivamente.

programa.py

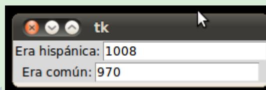
```
from Tkinter import *
ventana=Tk()
texto=StringVar()
texto.set("Escriba aquí su consulta...")
chisme=Entry(ventana,textvariable=texto)
chisme.pack()
ventana.mainloop()
```



Para organizar la disposición de los elementos que colocamos en la ventana contamos con el «widget» Frame:

programa.py

```
marco_hisp=Frame(ventana)
marco_hisp.pack()
marco_com=Frame(ventana)
marco_com.pack()
txt_hisp=Label(marco_hisp,text="Era hispánica:")
txt_hisp.pack(side="left")
txt_com=Label(marco_com,text="Era común:")
txt_com.pack(side="left")
n1=StringVar()
n1.set(1008)
n2=StringVar()
n2.set(970)
hueco1=Entry(marco_hisp,textvariable=n1)
hueco1.pack(side="left")
hueco2=Entry(marco_com,textvariable=n2)
hueco2.pack(side="left")
ventana.mainloop()
```



Por último, vamos a programar unas funciones para manejar los datos que el usuario introduzca en la ventana que acabamos de diseñar:

programa.py

```
def cambia_hisp(event):
    hisp=int(n1.get())
    if hisp>38:
        n2.set(hisp-38)
    elif hisp<1:
        n1.set(1008); n2.set(970)
    else:
        n2.set(hisp-39)
def cambia_com(event):
    com=int(n2.get())
    if com>0:
        n1.set(com+38)
    elif com==0 or com<-38:
        n1.set(1008); n2.set(970)
    else:
        n1.set(com+39)
(...)
hueco1.bind("<Return>",cambia_hisp)
hueco2.bind("<Return>",cambia_com)
hueco1.bind("<FocusOut>",cambia_hisp)
hueco2.bind("<FocusOut>",cambia_com)
```

A partir de aquí... práctica y consulta de manuales.

Conviene tener presentes los siguientes principios:

- Tener claro lo que se quiere conseguir con un programa
- Descomponer cada tarea en otras, cada vez más simples
- Documentar el código