Introducción a la programación de computadores

mediante el lenguaje Python

HOJAS DE AMPLIACIÓN

BLOQUES Y SANGRADO

En el lenguaje Python, el margen o sangrado de las líneas de un programa no es una cuestión estética, sino que viene determinado por la sintaxis del lenguaje. Un bloque está formado por varias instrucciones consecutivas, agrupadas de una de las dos maneras siguientes:

- Cada una en una línea y todas con el mismo sangrado.
- Todas en la misma línea, separadas con signos de punto y coma.

Las instrucciones de control que preceden un bloque (`if` (condición), `while` (condición)…) marcan con el signo de dos puntos el comienzo del bloque que rigen.

➠¿Cuál es la salida de los tres programas siguientes? Predice el resultado sin el ordenador y comprueba después tu respuesta.

```
for n in range(0,25):
    texto=str(n)
    if n%4==0:
        texto=texto+", múlt."
        print(texto)
```

```
for n in range(0,25):
    texto=str(n)
    if n%4==0:
        texto=texto+", múlt."
    print(texto)
```

```
for n in range(0,25):
    texto=str(n)
    if n%4==0:
        texto=texto+", múlt."
print(texto)
```

➠Los programas siguientes están mal escritos. Explica, en cada caso, por qué:

```
print("¡Hola! Escribe un número: ")
n=input()
if(n<0): print("Es negativo.")
    print("Hemos terminado.")
```

```
    print("¡Hola, mundo!")
```

```
n=input()
if(n==4)
   print("Cuatro")
```

```
i=1
while(i<10):
    cuadrado=i**2
    print((i,i**2))
        i=i+1
```

LA INSTRUCCIÓN «else»

Hemos visto cómo esta instrucción complementa de manera natural la bifurcación `if`:

```
if (condición):
    (bloque_1)
else:
    (bloque_2)
```

➠Completa las partes que faltan (BLOQUE_i) del siguiente programa para clasificar triángulos:

```
print("¿Cuánto miden los lados del triángulo que quiere clasificar?")
lados=list()
for i in range(0,3):
    lados.append(input())
lados.sort()
a=lados[0]; b=lados[1]; c=lados[2]
if(a+b<=c):
    print("No existe ningún triángulo con esos lados")
else:
    if(a**2+b**2>c**2):
        c1="acutángulo"
    elif(BLOQUE_1):
        c1="rectángulo"
    else:
        (BLOQUE_2)
    conj="y"
    if(a==b==c):
        c2="equilátero"
    elif(a!=b!=c):
        (BLOQUE_2)
    else:
        (BLOQUE_3)
    print("Es un triángulo "+c1+" "+conj+" "+c2+".")
```

En el lenguaje Python, la instrucción `else` también puede utilizarse con otras estructuras de control, como `for` o `while`. En estos dos casos, el bloque encabezado por `else` se ejecuta al final del bucle.

➠Estudia las parejas de programas siguientes. ¿Pueden devolver en algún caso resultados distintos?

```
a=input("Número: ")
for i in [2,3,5,7,11,13,17,19]:
    if a==i:
        print("Primo pequeño")
else:
    print("Búsqueda terminada")
```

```
a=input("Número: ")
for i in [2,3,5,7,11,13,17,19]:
    if a==i:
        print("Primo pequeño")
print("Búsqueda terminada")
```

```
a=input("Número: ")
for i in [2,3,5,7,11,13,17,19]:
    if a==i:
        print("Primo pequeño")
    elif a<i:
        print("Número compuesto")
        break
else:
    print("Búsqueda terminada")
```

```
a=input("Número: ")
for i in [2,3,5,7,11,13,17,19]:
    if a==i:
        print("Primo pequeño")
    elif a<i:
        print("Número compuesto")
        break
print("Búsqueda terminada")
```

➠Considera dos códigos con la siguiente estrucura:

```
while (condición):
    (bloque 1)
else:
    (bloque 2)
```

```
while (condición):
    (bloque 1)
(bloque 2)
```

Decide si cada una de las afirmaciones siguientes es verdadera o falsa:

1. Si `(bloque 1)` no contiene la instrucción `break`, ambos códigos son equivalentes.
2. Si `(bloque 2)` no contiene la instrucción `break`, ambos códigos son equivalentes.
3. Si `(bloque 1)` no contiene la instrucción `continue`, ambos códigos son equivalentes.
4. Si ni `(bloque 1)` ni `(bloque 2)` contienen la instrucción `continue`, ambos códigos son equivalentes.

LAS INSTRUCCIONES `break` Y `continue`

En principio, un programa consiste en una sucesión de operaciones, pero las instrucciones de control de flujo permiten una programación más rica, modificando el «camino» que sigue un programa en función de los resultados parciales que se van obteniendo. Hemos visto como los bucles `for` y `while` hacen que un proceso se repita para cada elemento de una lista o mientras se satisfaga una condición de control, respectivamente. En ocasiones es útil poder modificar el comportamiento normal de estos bucles.

Pongamos, por ejemplo, un programa que pretenda localizar a una persona con el apellido «Martínez» preguntando una por una:

```
for x in ["Pablo","Ana","Jimena","Rodrigo"]:
    ap=raw_input(x+", ¿cómo se apellida? ")
    if(ap=="Martínez"):
        print(x+" "+ap)
```

Si nos basta con localizar a una persona con ese apellido, no hay necesidad de seguir preguntando después del acierto, como haría el código superior. El programa siguiente interrumpe el bucle, ahorrando operaciones superfluas:

```
for x in ["Pablo","Ana","Jimena","Rodrigo"]:
    ap=raw_input(x+", ¿cómo se apellida? ")
    if(ap=="Martínez"):
        print(x+" "+ap)
        break
```

A diferencia de `break`, la función `continue` solamente interrumpe una iteración del bucle, pasando a la siguiente «vuelta» en lugar de concluir la ejecución de la instrucción `for` o `while` a la que corresponde.

➠Supongamos que queremos escribir un programa para evaluar la función

$$f(n) = \begin{cases} \dfrac{n^2-1}{6}, & \text{si } n \notin (2) \text{ y } n+1 \in (3) \\ \dfrac{n^2-1}{2}, & \text{si } n \notin (2) \text{ y } n+1 \notin (3) \\ n, & \text{si } n \in (2) \end{cases}$$

en $n = 0, \ldots, 10$ aprovechando el siguiente código:

```
n=0
f=list()
while n<=10:
    if n%2==0:
        f.append(n)
        n+=1
    aux=(n**2-1)/2
    if n%3==2:
        aux=aux/3
    f.append(aux)
    n+=1
for n in range(0,11):
    print("f("+str(n)+")="+str(f[n]))
```

¿Puede lograrse ese programa intercalando una sola instrucción `continue`?

«Tuplas» y listas

Python utiliza dos tipos de datos que permiten almacenar pares, ternas, y en general conjuntos **ordenados** con una cantidad arbitarira de elementos:

```
>>> t=123,"Tanos",True
>>> t
(123, 'Tanos', True)
>>> type(t)
<type 'tuple'>
>>> len(t)
3
```

```
>>> l=[123,"Tanos",True]
>>> l
[123, 'Tanos', True]
>>> type(l)
<type 'list'>
>>> len(l)
3
```

Una diferencia entre estos dos tipos radica en el carácter «inmutable» de las «tuplas» que, al igual que los números y las cadenas de texto, no pueden modificar su valor. Esto no impide que podamos asignar a una variable una «tupla» (o un número entero) y después otra distinta, borrando la anterior.

```
>>> t[0];t[2]
123
True
>>> t[1]="Cambio"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

```
>>> l[0];l[2]
123
True
>>> l[1]="Cambio"
>>> l
[123, 'Cambio', True]
```

Otra diferencia es que podemos añadir y eliminar elementos de una lista, utilizando «métodos» como `append`, `pop`, `remove` e instrucciones como `del`.

```
>>> lista=range(0,15)
>>> lista
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> lista.append("final")
>>> lista
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 'final']
>>> lista.pop()
'final'
>>> lista
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> lista.append(4);lista.append(5);lista
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 4, 5]
>>> lista.remove(4);lista
[0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 4, 5]
>>> lista.sort();lista
[0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> del lista[3:6];lista
[0, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Pilas y colas

En estas hojas trataremos de las *pilas* y las *colas*: dos estructuras de almacenamiento dinámico de datos cuyo funcionamiento es, en cierto modo, opuesto. Podemos entender ambas como sistemas para almacenar elementos que requieren ser procesados (por ejemplo, una lista de tareas pendientes). El término «almacenamiento **dinámico**» indica que la función que desempeñan este tipo de estructuras no se limita a almacenar un conjunto invariable de objetos. Los datos almacenados cambian a lo largo de la ejecución de un programa y la estructa ha de gestionar la manera de incluir nuevos datos y la de extraerlos para ser atendidos.

Los términos, bien descriptivos, de «pila» y «cola» hacen referencia a la prioridad que se asigna a cada elemento que ingresa a la estructura: en las colas, un nuevo proceso habrá de esperar a que sean ejecutados todos los que estaban almacenados antes; mientras que en las pilas, por el contrario, se le da la máxima prioridad al proceso entrante. Basta evocar la cola de un cine y una pila de platos para ejemplificar estos dos modos de organización de tareas.

En Economía, se suele hacer referencia a este paradigma mediante las siglas inglesas *FIFO (first in, first out)* y *LIFO (last in, first out)*. Así por ejemplo, en el contexto de la tributación fiscal derivada de un incremento pratrimonial causado por la compraventa de acciones, es de relevancia la interpretación que se hace de qué acciones se están transmitiendo. Por ejemplo, supongamos que un contribuyente realiza los movimientos reflejados a continuación sobre las acciones de una misma compañía:

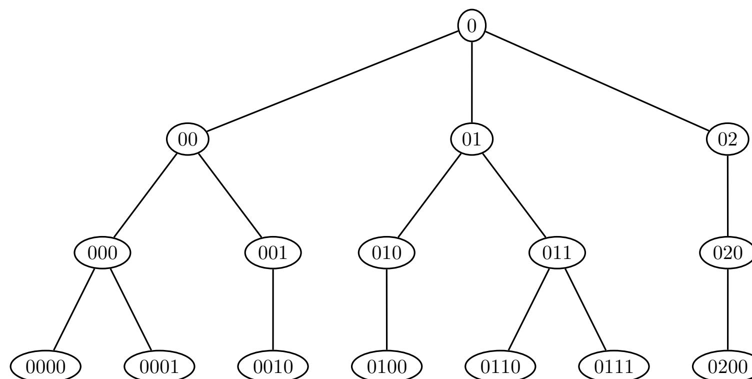
2 de enero	COMPRA	1000 acciones	0'5€/acción	500€
3 de febrero	COMPRA	1000 acciones	0'9€/acción	900€
4 de marzo	VENTA	1000 acciones	1€/acción	1000€

A la hora de rendir cuentas con el fisco, es diferente seguir el criterio FIFO, según el cual el incremento patrimonial ha sido de 500€; que el LIFO, con el que solo habría de declararse una ganancia de 100€.

En una primera aproximación, la organización de tareas en «cola», siguiendo el modelo FIFO, parece más razonable y resulta sencillo localizar ejemplos donde aplicarla. Así por ejemplo, del encargado de una biblioteca que reciba tarjetas de los usuarios solicitando volúmenes, se espera que dé curso a estas peticiones según su orden de recepción. Tampoco se considera apropiado que alguien se cuele[1] en la panadería o en la oficina de la Seguridad Social.

No obstante, tampoco escasean las situaciones en que surge naturalmente la estructura de «pila». En un contexto en que sea preferible —o incluso necesario— ir concluyendo tareas parciales a respetar el orden en que se van prescribiendo, resulta más conveniente la estructuración de los procesos en una pila.

Así por ejemplo, la curiosidad inherente al hombre hace más habitual el recorrido de los nodos de una estructura de árbol en profundidad que su «explosión» por niveles. Por otra parte, esta segunda opción requiere en general una ocupación de memoria significativamente mayor.



Si consideramos el árbol de la figura, una búsqueda exhaustiva en profundidad, utilizando una estructura de pila para recordar los nodos por explorar requiere almacenar simultáneamente un máximo de 4 entradas. En cambio, utilizando una estructura de cola se necesitarían 7 posiciones de memoria.

[1]Según el D.R.A.E., las etimologías de «colarse» y «cola» no están relacionadas.

LIFO				FIFO						
0				0						
00	01	02		00	01	02				
00	01	020		01	02	000	001			
00	01	0200		02	000	001	010	011		
00	01			000	001	010	011	020		
00	010	011		001	010	011	020	0000	0001	
00	010	0110	0111	010	011	020	0000	0001	0010	
00	010	0110		011	020	0000	0001	0010	0100	
00	010			020	0000	0001	0010	0100	0110	0111
00	0100			0000	0001	0010	0100	0110	0111	0200
00				0001	0010	0100	0110	0111	0200	
000	001			0010	0100	0110	0111	0200		
000	0010			0100	0110	0111	0200			
000				0110	0111	0200				
0000	0001			0111	0200					
0000				0200						

Encontramos otro ejemplo al desarrollar un algoritmo para el problema conocido como «de las torres de Hanoi». Así, la tarea «mover la torre de la posición 1 a la 2» se descompone en las tres siguientes:

1. Mover la torre salvo su base de la posición 1 a la 3.
2. Mover la base de la posición 1 a la 2.
3. Mover la torre situada en la posición 3 a la 2.

Es necesario concluir cada paso antes de comenzar el siguiente; lo que impide, por ejemplo, que los procesos segundo y tercero se ejecuten antes de cualquier subproceso que pida la ejecución del primero. Así, al ir desarrollando estos procesos, los pasos que se vayan prescribiendo han de priorizarse a los ya almacenados. Esto hace inviable una estructura de cola.

Pueden obtenerse otros ejemplos de lenguajes de programación como PostScript o una calculadora que acepte entradas escritas en notación polaca[2] inversa. En PostScript, la pila es la estructura de datos predominante. Los comandos, diccionarios, entornos gráficos, etc. se organizan según esta estructura. Por ejemplo, al invocar el comando «suma», el intérprete busca los dos últimos valores almacenados en la pila de operandos, comprueba que son números, y los suma en caso afirmativo, colocando el resultado en esa misma pila.

Ejercicio: *Escribe un programa en Python que calcule una lista ordenada de movimientos que resuelva el «Problema de las Torres de Hanoi» de orden n. Utiliza una variable de tipo* `list`*, junto con los métodos* `append` *y* `pop`*; y no otras, de modo que la estructura de datos utilizada se corresponda con una pila.*

[2]La llamada notación polaca (p.ej, la expresión «+ 4 5» se evaluaría en 9) recibe su nombre del lógico polaco Jan Łukasiewicz. Esta notación permite prescindir de paréntesis y requiere que cada comando defina unívocamente la cantidad de argumentos que precisa.