Listas y ciclos `for-in`

Anteriormente hemos tratado con distintos tipos de datos que Python ofrece, como el entero (`int`), el entero de largo (`long`), los números de punto flotante (`float`) y los valores booleanos (`boolean`), sin embargo todos estos tipos comparten una característica en común: Son tipos de datos **atómicos** (estructuralmente indivisibles). Esto significa que cuando hacemos una asignación, la variable almacena sólo un valor.

Ejemplo 1

```
>>> flotante = 2.5
>>> flotante
2.5
>>> enteroLargo = 3L
>>> enteroLargo
3L
>>> verdad = False
>>> verdad
False
>>>
```

Sin embargo, en el mundo real existen ejemplos de **conceptos asociados a varios valores**. Por ejemplo “estudiantes del curso”, “notas del estudiante”, “nombres de los miembros del club”, etc.

En consecuencia, al programar una solución a un problema real, muchas veces necesitamos asociar **varios datos** a un **mismo nombre**. Supongamos, por ejemplo, que se nos pide un programa que calcule el promedio y la desviación estándar de la estatura de los alumnos que están cursando Fundamentos de Computación y Programación. ¡Podríamos necesitar 1.600 variables!

Para facilitar el manejo de datos en problemas como éste, Python permite el manejo de **listas** de elementos, en las cuáles se pueden almacenar **varios datos** dentro de **una sola variable**. Veamos cómo funcionan las listas en el siguiente ejemplo.

Ejemplo 2

```
>>> [1, 2, 3, 4, 5, 6.0]
[1, 2, 3, 4, 5, 6.0]
>>> lista1 = [1, 1+1, int(6/2.0)]
>>> lista1
[1, 2, 3]
>>> listaVacia = []
>>> listaVacia
[]
```

Podemos ver que para definir una lista en Python, simplemente colocamos los **elementos de la lista** dentro de **paréntesis cuadrados** (`[ ]`), separándolos por **comas** (`,`). Podemos guardar la lista en una variable para usarla posteriormente en el programa. En resumen la sintaxis es la siguiente:

Importante

```
variable =  [<elemento_1>, <elemento_2>, ... , <elemento_n>]
```

Ahora podemos guardar muchos valores en una única variable de **tipo lista** (`list` en Python). Pero como los valores no sólo son para guardarlos solamente, sino que también necesitamos **operar con ellos**, requerimos formas de accederlos. Observemos el siguiente código en Python:

Cuadrados.py

```
numeros = [2, 5, 7, 13]
print numeros

numeros[0] = numeros [0]**2
numeros[1] = numeros [1]**2
numeros[2] = numeros [2]**2
numeros[3] = numeros [3]**2

print numeros
```

Pregunta 1

Trabaja ahora con tu grupo en responder la pregunta 1 de la actividad.

Ahora, es claro que el programa `Cuadrados.py` eleva los elementos de la lista `numeros` al cuadrado. Podemos notar que es posible referirse a **un elemento** de la lista en particular con la **posición** que tiene en la lista. Podemos ver que la lista `numeros` tiene cuatro elementos (**largo** 4), y que sus elementos están **indexados** con las posiciones del 0 al 3. Para acceder al primer elemento, se escribe el nombre de la lista con la posición 0 en paréntesis cuadrados: `numeros[0]`. Para obtener el número 7 en la lista, en la penúltima posición de la lista, tenemos referenciarlo escribiendo: `numeros[2]`. El último elemento de la lista será `numeros[3]`. La figura 1 resume esta **indexación posicional**.

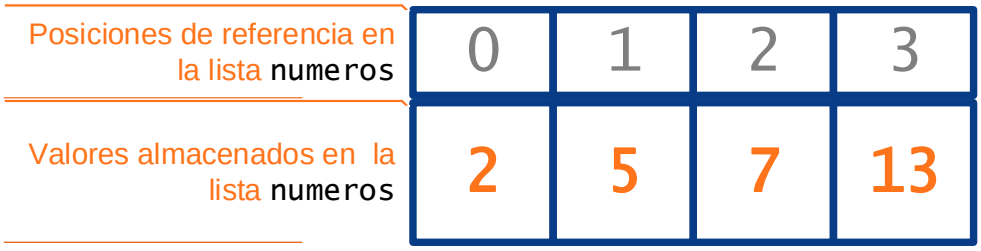

Figura 1. Posiciones y valores en la lista `numeros`

Pero también existen operaciones y funciones que podemos aplicar a una **lista completa**, como una sola entidad. Por ejemplo, podemos obtener el **largo de una lista** utilizando la función nativa de Python `len()`.

Ejemplo 3

```
>>> len([1, 2, 3, 4, 5, 6.0])
6
>>> lista1 = [1, 1+1, int(6/2.0)]
>>> len(lista1)
3
>>> listaVacia = []
>>> len(listaVacia)
0
```

También tenemos el operador **concatenación** (+) que nos permite unir dos listas.

Ejemplo 4

```
>>> lista = [1, 2, 3]
>>> lista
[1, 2, 3]
>>> lista = lista + [4, 5]
>>> lista
[1, 2, 3, 4, 5]
>>>
```

Pregunta 2

Trabajen ahora en la segunda pregunta de la actividad.

Las listas son **naturalmente iterativas**, ya que muchos problemas en el mundo real requieren aplicar alguna operación a cada elemento de una lista. Por ejemplo, calcular el promedio de **cada alumno del curso**, calcular el impuesto a pagar por **cada funcionario de la Universidad**, etc.

Estructura `for-in`

Para realizar iteraciones en Python, hemos visto que es posible utilizar la sentencia `while`. Por ejemplo, si quisiéramos obtener las potencias de 0 a 7 de 2 por pantalla para trabajar en ejercicios con base binaria, deberíamos escribir algo similar a esto:

PotenciasDeDos.py

```
# Definición de los exponentes a utilizar
exponentes = [0, 1, 2, 3, 4, 5, 6, 7]

# Muestra la potencia de 2 para cada exponente
posicion = 0
while posicion < len(exponentes):
    print posicion, 2 ** exponentes[posicion]
    posicion = posicion + 1
```

Sin embargo, se obtiene una solución mucho **más natural** cuando se aprovecha la naturaleza iterativa de las listas a través de la **sentencia `for-in`**:

PotenciasDeDosNatural.py

```
# Muestra la potencia de 2 para los exponentes del cero al siete
for exponente in [0, 1, 2, 3, 4, 5, 6, 7]:
    print exponente, 2 ** exponente
```

Sabiendo un poquito de inglés, podemos notar lo natural que esta sintaxis resulta: "*para cada exponente en la secuencia [0, 1, 2, 3, 4, 5, 6, 7: mostrar el exponente y 2 elevado al exponente*". La sintaxis en Python para un ciclo `for-in` es la siguiente:

Importante

for <identificador> **in** <lista de elementos>**:**
    <operaciones a realizar>

Pregunta 3

Es hora de desarrollar la pregunta 3 de la actividad de hoy.

`For-in` en Python y `for` en otros lenguajes

En otros lenguajes de programación existe la sentencia for, sin embargo esta es un poco distinta al for-in de Python, pues mientras aquí la sentencia se lee:

- Para cada elemento de la secuencia haga : <instrucciones>

```
for <elemento> in <listado de elementos>:
```

Por ejemplo: `for elemento in [1, 2, 3, ,4 ,5 ,6 ,7]:`

En otros lenguajes de programación se entiende como:

- Para cada iteración desde <inicio> hasta <condición de fin>, con paso de <incremento> haga {<instrucciones>}

```
for(<inicio>, <condición de fin>, <incremento>){…
```

Por ejemplo: `for(i = 0; i < 7, i ++){…`

Es decir, mientras en Python limitamos la iteración de for-in para cada elemento de la lista, la sentencia for de otros lenguajes limitan las iteraciones completamente, al decirle explícitamente al programa dónde empezar, hasta donde llegar y que tan largos deben ser los pasos a dar.

Construyendo listas rápidamente

La función nativa `range(<desde>, <hasta>, <incremento>)` es muy utilizada en Python para iterar un bloque de código un **número fijo de veces** que es conocido *a priori*. La función crea listas de números enteros de acuerdo a los tres parámetros que se le entregan. El parámetro `<desde>` indica el **valor inicial** de la secuencia a generar (el primer elemento). Éste es un **parámetro opcional**, que si se omite asume valor **cero**. El parámetro `<hasta>` es obligatorio y corresponde al **límite superior** de la secuencia. Debe tenerse en cuenta que el último elemento en la secuencia es siempre **menor que** el valor de este parámetro. Por último, el parámetro opcional `<incremento>` especifica la **diferencia aritmética** que existe entre cada elemento consecutivo de la secuencia. También es un parámetro opcional y si se omite se asume el valor **uno**.

Los siguientes son ejemplos del funcionamiento de la función `range()`. Observémoslos con cuidado para entender por qué se obtiene la secuencia resultante.

Ejemplo 5

```
>>>range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 10, 3)
[1, 4, 7]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(10, 3)
[]
>>> range(5, 10, 10)
[5]
>>>
```

La función `range()` nos sirve para crear rápidamente listas para nuestros ciclos `for-in`. Por ejemplo, si en el programa `PotenciasDeDosNatural.py` necesitáramos ahora las 30 primeras potencias de 2, resultaría engorroso definir una lista con los números del 0 al 30. Más simple es utilizar la función `range()` para generarla.

PotenciasDeDosNaturalConRange.py

```
for exponente in range(0, 30):
    print exponente, 2 ** exponente

# Equivalente a:

exponentes = range(0, 30)
for e in exponentes:
    print e, 2 ** e
```

Pregunta 4

Trabajemos ahora en la cuarta pregunta de nuestro apunte de hoy.

Manejando texto

En Python hemos visto que a menudo entregamos mensajes utilizando palabras u oraciones entre comillas, como por ejemplo `"HOLA MUNDO"`. Este es un **tipo de dato** conocido como **string** (**`str`** en Python) que la mayoría de los lenguajes de programación incluye.

Al igual que las listas, el **string** es un tipo de dato **compuesto** (no atómico), en este caso, exclusivamente por **caracteres** (**chr** en Python). Por esta razón, podemos iterar sobre los caracteres que contiene y podemos concatenarlos con otros strings.

Ejemplo 6

```
>>> for c in "Hola Mundo":
      print c

H
o
l
a

M
u
n
d
o
>>>
>>> "Hola" + " " + "Mundo"
'Hola Mundo'
>>>
```

Sin embargo, a diferencia de las listas, un string es **inmutable**, es decir no podemos cambiar alguno de los elementos que lo componen.

Ejemplo 7

```
>>> saludo = "Hola Mundo"
>>> saludo[0] = 'h'

Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    saludo[0] = 'h'
TypeError: 'str' object does not support item assignment
>>>
```

Con esta restricción en mente, podemos hacer operaciones iterativas sobre texto, como por ejemplo, contar cuántas palabras contiene un string.

ContarPalabras.py

```
# Entrada
texto =  input("Ingrese texto entre comillas: ")

# Procesamiento
nEspacios = 0
for caracter in texto:
  if caracter == ' ':
    nEspacios = nEspacios + 1

# Salida
print "Su texto contiene", nEspacios + 1, "palabras"
```

Pregunta 5

Para terminar resuelve con tus compañeros la pregunta 5 de la actividad