



Listas

- La **lista** es un tipo de colección ordenada. Sería equivalente a lo que en otros lenguajes se conoce por **arrays, o vectores**.
- Python utiliza listas para aquellas estructuras tipo tabla que son dinámicas (pueden cambiar de número de elementos).
- La definición de una lista se realiza encerrando entre corchetes los valores que necesitamos almacenar, siendo estos de cualquier tipo posible , incluso otras tablas, diccionarios, etc.

lVal1 = ["valor", 2, 2 + 4j]

- El acceso y modificación del contenido de una de las posiciones se realizará por un **índice numérico**, empezando dichos valores en **cero**.

- 
- Los **índices** en las listas de Python no tienen por qué ser solo positivos, se pueden utilizar valores **negativos** indicando que se empieza a **contar desde el final**. La posición `lVal1[-1]` sería la última, -2 la antepenúltima y así sucesivamente.
 - Conseguir varios elementos consecutivos de la lista como una nueva es un proceso muy sencillo, indicando los índices inicial y final a conseguir (el final no se incluye, es decir, se excluye) separándolos por dos puntos (:).
 - Este mecanismo también permite utilizar números negativos para indicar la posición inicial y final, así como no establecer cualquier índice, si deseamos coger desde el primero o hasta el último elemento.

- 
- Para las listas se definen solo dos **operadores**, el operador **concatenación (+)** y el operación de **repetición (*)** similar a los diccionarios.
 - Las listas son ***dinámicas***, por lo que puede variar el número de elementos que la constituye durante la ejecución del programa. Para implementar este mecanismo se han dotado al tipo de una serie de métodos que lo facilitan.
 - Para **aumentar los componentes** utilizaremos uno de los tres **métodos** existentes para tal fin:
 - **append(valor)** añade el valor al final de la tabla incrementando el último índice.
 - **insert(posición,valor)** inserta en una posición un valor, desplazando el resto.
 - **extend(iterable)** concatena la lista actual a la que se le pasa creando una nueva con todos los valores, este método implementa la misma funcionalidad que el operador concatenación (+) pero de una forma mucho más eficiente.



➤ También tenemos varios **métodos para disminuir los componentes**:

- **pop(valor_opcional)** extraerá el último de la lista si no se proporciona un índice o el elemento del índice indicado, similar a: **del lVal[indice]**.
- **remove(valor)** quitará el primero que concuerde con el valor, produciéndose un error en caso de que no exista coincidencia.

- Una lista **sí tiene orden**, el que se encuentra al recorrer la lista desde la posición inicial hasta la final de forma ascendente.
- Como el orden es importante tenemos un par de mecanismos para realizar búsquedas sobre los contenidos. El primero es el método **index(valor)** que nos devolverá la posición en la lista del valor pasado. El segundo es el uso de la palabra clave **in** para determinar si un valor se encuentra en alguna posición ("**c**" **in** **IVal1**).
- Podemos aplicar una operación (un mapeo) a cada elemento de la lista a la vez que determinamos si dicho componente formará parte de la nueva lista. La estructura tiene la siguiente sintaxis: **[operación for variable in lista condición_opcional]**. Cuando veamos los bucles, veremos ejemplos de uso.

Actividad 5.13

Crea un proyecto llamado **Listas** y un módulo **Actividad5-13** con el código siguiente. Ejecútalo.

```
# -*- coding: utf-8 -*-
#Definición
IVall = ["valor1", 2, 2 + 4j]
print "Lista:", IVall
print "Lista valor con índice 2 d[2]:", IVall[2]

#Es dinámica
IVall.append(6 + 7j)
print "Lista añadir:", IVall
IVall[3] = "Mi cadena"
print "Lista modificar:", IVall
del IVall[0] # Base cero
print "Lista borrar:", IVall

#Se puede conseguir partes
print "Desde el final índice negativo", IVall[-2] # -1 es el último
print "Una parte", IVall[1:] # desde el segundo hasta el final, empezamos en cero
print "Una parte", IVall[1:2] # inicio incluido índice fin excluido

#Operadores +, +=, *
print "Dos listas:", IVall + IVall
print "Una lista tres veces:", IVall * 3
IVall2 = [True]
IVall2 += IVall
print "+=", IVall2


#Funciones básicas
print " Funciones básicas "
print IVall
print len(IVall)
print IVall.index(2 + 4j) #índice del valor 2+4j
print 3 in IVall # ¿3 está en la lista?
print IVall.remove(2) # borramos por valor, no por índice
print IVall
print IVall.pop() # Extrae el último, no tiene parámetro
print IVall
print IVall.insert(len(IVall), "45") # Insertamos en la posición final
print IVall
print IVall.extend([1, 2, 3]) # Insertamos todos los del objeto iterable (en este caso la lista [1,2,3])
print IVall
```

Actividad 5.13 (...continuación)

```
Console
<terminated> C:\Users\Web\workspace>Listas\src\Actividad5-13\Actividad5-13.py
Lista: ['valor1', 2, (2+4j)]
Lista valor con índice 2 d[2]: (2+4j)
Lista añadir: ['valor1', 2, (2+4j), (6+7j)]
Lista modificar: ['valor1', 2, (2+4j), 'Mi cadena']
Lista borrar: [2, (2+4j), 'Mi cadena']
Desde el final índice negativo (2+4j)
Una parte [(2+4j), 'Mi cadena']
Una parte [(2+4j)]
Dos listas: [2, (2+4j), 'Mi cadena', 2, (2+4j), 'Mi cadena']
Una lista tres veces: [2, (2+4j), 'Mi cadena', 2, (2+4j), 'Mi cadena', 2, (2+4j), 'Mi cadena']
+= [True, 2, (2+4j), 'Mi cadena']
Funciones básicas
[2, (2+4j), 'Mi cadena']
3
1
False
None
[(2+4j), 'Mi cadena']
Mi cadena
[(2+4j)]
None
[(2+4j), '45']
None
[(2+4j), '45', 1, 2, 3]
```

Tuplas

- Las **tuplas** se interpretan como **listas inmutables** tanto en tamaño como en contenido. Esta función permite que sean más ligeras que las listas y mucho más eficientes, a cambio de ser invariables.
- Para definir una **tupla** cambiaremos los corchetes de las listas por paréntesis (**tVal1 = ("valor1", 2, 2 + 4j)**) y accederemos de la misma manera que las listas, usando los corchetes para tal fin (**tVal1[2]**) evitando usar una asignación ya que no está permitido (**tVal1[2]=3** daría un error). Otra característica que comparte con las listas es el uso de los índices para seleccionar un elemento o un rango (índices negativos, uso de los dos puntos, base cero, etc.)

- 
- Las **tuplas**, al ser objetos fijos, no tienen los métodos que permiten variar el contenido o el tamaño existentes en las listas, solo está presente el método **index(valor)** que devolverá la posición del elemento que concuerde con el valor pasado.
 - Las listas y las **tuplas** son tipos de datos similares en cuanto funcionamiento, por lo que es factible crear una a partir de la otra. Así, si tenemos una lista con la llamada a la función **tuple(var_lista)** formaremos una **tupla** desde la lista pasada como parámetro. Si necesitamos una lista a partir de una **tupla**, llamaremos a la función **list(var_tupla)** con la que conseguiremos una lista, modificable por tanto, desde una **tupla** ya existente.

Actividad 5.14

Crea un proyecto llamado Tuplas y un módulo Actividad5-14 con el código

```
#!/usr/bin/env python3
# -*- coding:UTF-8 -*-
#Definición
tVal1=("Valor1",2,2+4j)
print "Tupla:", tVal1
print "Tupla valor con índice 2 d[2]:", tVal1[2]

#No Es dinámico
#Se puede conseguir partes
print "Desde el final índice negativo", tVal1[-2] # -1 es el último
print "Una parte", tVal1[1:]
print "Una parte", tVal1 [1:2] # inicio incluido índice fin excluido

#Operadores +,+=,*
print "Dos tuplas:", tVal1 + tVal1
print "Una tupla tres veces:", tVal1 * 3

#Funciones básicas
print "Funciones básicas"
print tVal1
print len(tVal1)
print tVal1.index(2 + 4j)
print 3 in tVal1 # ¿Valor 3 en la tupla?
```

Console

```
<terminated> C:\Users\Web\workspace\Tuplas\src\Actividad5-14\Actividad5-14.py
Tupla: ('Valor1', 2, (2+4j))
Tupla valor con índice 2 d[2]: (2+4j)
Desde el final índice negativo 2
Una parte (2, (2+4j))
Una parte (2,)
Dos tuplas: ('Valor1', 2, (2+4j), 'Valor1', 2, (2+4j))
Una tupla tres veces: ('Valor1', 2, (2+4j), 'Valor1', 2, (2+4j), 'Valor1', 2, (2+4j))
Funciones básicas
('Valor1', 2, (2+4j))
3
2
False
```

Variables

- No es obligatorio definir una variable para su uso. Cuando requiramos utilizar una variable nueva, simplemente le asignamos el valor que deseemos y desde ese punto del programa estará accesible.
- Si bien no es imprescindible su definición, es imperativo inicializar la variable antes de usarla. El lenguaje permite inicialmente asignar a la variable (**a=23**) un valor numérico y posteriormente en el código otro diferente, por ejemplo una cadena de texto (**a="Hola"**) sin ningún tipo de problema, será el intérprete el que en tiempo de ejecución determine el tipo.
- Podemos definir dos tipos de variables según el ámbito en el que estén visibles: **globales y locales**.

- Las primeras tienen valor a lo largo de todo el programa, funciones y clases que definamos. Las segundas solo tienen sentido dentro del bloque en el que se haya inicializado, como son dentro de las funciones. Es factible usar el mismo nombre en una variable **global** y en una **local**, el efecto es que la variable **local** enmascarará la **global**, siendo solo accesible esta.
- **No está permitido la asignación en línea** tipo $(a=b=c=9)$, pero usando las **tuplas** podemos realizar una función similar. Las **tuplas** permiten asignarse entre sí, siendo factible que en la parte izquierda de una asignación aparezca una tupla de n elementos y en la parte derecha de la asignación otra tupla de también n elementos. El resultado que se produce es que a cada elemento de la tupla izquierda se le asigna el valor correspondiente de la derecha **$((v1, v2, v3) = (1, "ABC", True), v1$ valdrá **1**, $v2$ será **"ABC"** y $v3$ contendrá **True**)**.

- Para imprimir una variable por pantalla con un formato usaremos una cadena de formato. A la función **print** (mostrar en pantalla) le pasamos una cadena que especificará cómo deseamos imprimir las variables indicando la posición, tipo y formato de cada una con una serie de caracteres de formato y las variables a utilizar.
- La sintaxis de la cadena de formato usa expresiones tipo %s, %d, %f para representar el tipo. Para separar la cadena de formato de los valores a usar se utiliza el carácter tanto por ciento (%), encerrando los valores a usar dentro de una **tupla**.
- Al usar **print**, si separamos los valores por comas (,) se imprimirá un espacio entre ellos, si usamos el operador concatenación (+) tendremos que añadir nosotros el espacio donde sea necesario y es obligatorio convertir a cadena a través de la función **str()** toda variable que no sea de ese tipo.

- Un problema que nos encontramos es la utilización de los caracteres de acentuado y la ñ. Aunque el código fuente se está escribiendo en formato UTF-8, la función de entrada **raw_input()** se configura de manera general, siendo imprescindible codificar los caracteres al lenguaje adecuado antes de imprimirlos usando la función **encode(norma_ISO)** para una correcta visualización en la consola.
- **Python** añade un conjunto de variables del sistema que sería deseable conocer (véase <http://docs.python.org/2/library/sys.html>). Nos vamos a centrar en las que se utilizan para recoger los parámetros que nos pasa un usuario cuando ejecuta nuestro script. En **Python** se ha seguido una nomenclatura similar a C, creando una lista llamada **argv** que nos dará acceso a todos los parámetros que nos pasen desde el sistema, siempre que antes hayamos importado el módulo SYS (**import sys**).

Actividad 5.15

Crea un proyecto llamado **Variables** y un módulo Actividad5-15 con el

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# Definición sin tipo al asignar, pero obligatorio asignar antes de usar

sVal1 = "Valor"
sVal2 = "Mi cadena \
    Larga pero en varias líneas"
print sVal1, "-----", sVal2
tTupla1 = (1, "ABC", True)
(v1,v2,v3) = tTupla1
print "Variables v1 ,v2 y v3:", v1, v2, v3

# Cadenas de formato %d,%s,%f
iVal1 = 23
sVal2 = "José"
print "%s tiene %d años" % (sVal2, iVal1)
print "%s tiene %.2f años" % (sVal2, iVal1)

# Entrada de datos
nombre = raw_input("Cómo te llamas?")
print "Encantado," + nombre # Comprobar la necesidad del espacio al usar
```

Console

<terminated> C:\Users\Web\workspace\Variables\src\Actividad5-15\Actividad5-15.py

Valor ----- Mi cadena Larga pero en varias líneas

Variables v1 ,v2 y v3: 1 ABC True

José tiene 23 años

José tiene 23.00 años

Cómo te llamas?Teresa

Encantado,Teresa

Control del flujo

- Para controlar el flujo de nuestros programas disponemos de la **sentencias condicionales** y de los **bucles**.
- Las **sentencias condicionales** nos permiten ejecutar distintos fragmentos de código dependiendo de unas condiciones. Los **bucles** nos permiten ejecutar un fragmento de código un número determinado de veces mientras se cumpla una determinada condición.
- Recordemos que no existen elementos para determinar el final de un bloque. Para establecer el contenido se deberá sangrar en el mismo número de espacios todas aquellas líneas que formen parte de él, añadiendo diferentes niveles de sangrado según los bloques que necesitemos crear. Para abandonar un bloque se eliminará dicho sangrado alineándolo con la sentencia que ha creado el bloque.

Sentencias condicionales

- Sirven para hacer que se ejecute un fragmento de código o no en función de una condición.

- **if**

- Ejecuta un fragmento de código si se cumple una condición. Ejemplo:

```
if tipo == "socio ":  
    print "Acceso permitido"  
    print "Que tenga un buen día"
```

- **if...else**

- Ejecuta un bloque de código si se cumple una condición y otro en caso contrario.

Ejemplo:

```
if tipo == "socio":
```

```
    print "Acceso permitido"
```

```
    print "Que tenga un buen día"
```

```
else:
```

```
    print "Acceso denegado"
```

```
    print "Lo siento"
```

- **if..elif...elif...else**

➤ Se usa cuando necesitamos controlar más de dos casos de una condición. Ejemplo:

```
if numero < 0:  
    print "Negativo"  
elif numero > 0:  
    print "Positivo"  
else:  
    print "Cero"
```

➤ También disponemos de una estructura if...else en una única línea: Ejemplo:

```
variable = "par" if (num % 2 == 0) else "impar"
```

Actividad 5.16

Crea un proyecto llamado **Control de flujo** y un módulo **Actividad5-16** con el código siguiente. Ejecútalo.

```
#-*-coding: utf-8 -*-

#Bloque if
sTipo = "nosocio"
if sTipo == "socio":
    print "Acceso permitido"
    print "Que tenga un buen día"

#Bloque if...else
if sTipo == "socio":
    print "Acceso permitido"
    print "Que tenga un buen día"
else:
    print "Acceso denegado"
    print "Lo siento"

#Bloque if...elif...elif...else
iNumero=5
if iNumero < 0:
    print "Negativo"
elif iNumero > 0:
    print "Positivo"
else:
    print "Cero"

#En una línea
iVal1=3
print "cero" if (iVal1 == 0) else "Distinto de cero"
```

Console

```
<terminated> C:\Users\Web\workspace\Control de flujo\src\Actividad5-16\Actividad5-16.py
Que tenga un buen día
Acceso permitido
Que tenga un buen día
Positivo
Distinto de cero
```

Bucles

- Los bucles nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.

- **while**

- El bucle **while** ejecuta un fragmento de código mientras se cumpla una condición.

```
variable = 0
```

```
while variable < 10:
```

```
    variable = variable + 1
```

```
    print "El número a mostrar es " + str(variable )
```

- Dos **instrucciones** que nos podemos encontrar en los bucles son:
 - **break:** termina el bucle y continúa en la siguiente instrucción fuera del bloque
 - **continue:** pasa directamente a la siguiente iteración del bucle.
- Ejemplo: bucle infinito que va repitiendo todo lo que el usuario ha introducido por teclado hasta que escribe «salir», entonces se sale del bucle.

```
entrada= ""
```

```
while True:
```

```
    entrada = raw_input("> ")
```

```
    if entrada == "salir":
```

```
        break
```

```
    else:
```

```
        print entrada
```

- Ejemplo: escribir los números impares menores que 20. Cuando el resto de la división del número entre 2 es 0 (que indica que se trata de un número par), salta a la siguiente iteración del bucle sin imprimir el valor del número.

```
numero=0
```

```
while numero < 20:
```

```
    numero = numero + 1
```

```
    if numero % 2 == 0:
```

```
        continue
```

```
    print "Número: " + str(numero)
```

- **for..in**

➤ Para recorrer un objeto iterable (lista, tupla, etc.) se utiliza la estructura


for varElemento in objetoIterable.

➤ El bucle recorrerá cada uno de los componentes de la lista asignando a la variable uno de ellos en cada vuelta y realizando las acciones que definamos dentro del bloque correspondiente. Ejemplo:

```
laborables= ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes"]
```

```
for elemento in laborables:
```

```
    print elemento
```


- 
- Aunque **Python** carece de bucles tipo **C** (*for(;;)*) se pueden simular fácilmente creando una lista con los valores numéricos deseados y posteriormente recorrer dicha lista con la estructura presentada.
 - Para formar la lista se debe utilizar la función **range(inicio, fin, [salto])** creando el conjunto de números a recorrer en vez de hacerlo a mano.
 - Una construcción del bucle **for** junto con la sentencia **if** nos permite filtrar rápidamente listas: **[elemento for elemento in lista if len(elemento) > 1]** generando una nueva lista con los objetos que cumplan la condición. En este caso con aquellos cuya longitud sea mayor de uno.

Actividad 5.17

En el proyecto llamado **Control de flujo** crea un módulo **Actividad5-17** con el código siguiente. Ejecútalo.

```
# -*- coding: utf-8 -*-
#while
entrada=""
while True:
    entrada = raw_input(">")
    if entrada == "salir":
        break
    else:
        print entrada
#for
tVall =(1,2,3,4,5)
for ele in tVall:
    if ele % 2 == 0: print str(ele) + " es par"
#Simular bucle for vueltas fijas
for ele in range(5, 25):
    print "Valor:", ele
```

```
>hola
hola
>adios
adios
>salir
2 es par
4 es par
Valor: 5
Valor: 6
Valor: 7
Valor: 8
Valor: 9
Valor: 10
Valor: 11
Valor: 12
Valor: 13
Valor: 14
Valor: 15
Valor: 16
Valor: 17
Valor: 18
Valor: 19
Valor: 20
Valor: 21
Valor: 22
Valor: 23
Valor: 24
```

Actividades Repaso

1. Escribe un programa que pida un número (input) y determine si es par o impar, mostrando el mensaje correspondiente
2. Escribe un programa que determine cuál de tres números introducidos por el usuario es mayor.
3. Escribe un programa que permita imprimir los números del 1 al 100 y calcular la suma de todos los números pares por un lado, y por otro, la de los impares.
4. Escribe un programa que lea valores del usuario hasta que teclee un número par, utilizando un bucle while.
5. Escribe un programa que lea una cadena de texto del usuario y para cada letra indique si es una vocal o una consonante