# Módulo de adaptación

## Master en Business Intelligence y Big Data

**PROFESOR/A**
Antonio Sarasa Cabezuelo

Escuela de organización industrial

www.eoi.es

# Introducción a Python(II)

# Índice

# Ficheros

- Cuando se desea leer o escribir en un archivo primero se debe abrir el fichero usando la función open().
- Si la apertura tiene éxito, el sistema operativo nos devuelve un manejador de fichero (file handle) que se puede utilizar para leer los datos. Sólo se obtiene el manejador si el fichero especificado existe y además se dispone de los permisos apropiados para poder leerlo. Si el fichero no existe, open fallará y no se obtiene ningún manejador para poder acceder a su contenido.

# Ficheros

- Un fichero de texto puede ser considerado una secuencia de líneas. Para dividir el archivo en líneas, existe un carácter especial que representa el “final de línea”, llamado salto de línea (newline). El carácter salto de línea se representa por barra invertida-n en las cadenas. A pesar de que parezcan dos caracteres, se trata en realidad de uno sólo.

```
In [1]: cosa = '¡Hola\nMundo!'
        cosa

Out[1]: '\xc2\xa1Hola\nMundo!'

In [2]: print cosa

        ¡Hola
        Mundo!

In [3]: len(cosa)

Out[3]: 13
```

# Ficheros

- Por tanto en las líneas de un fichero existe un carácter especial invisible llamado salto de línea al final de cada línea, que marca donde termina la misma y comienza la siguiente.

# Ficheros

- Una vez que se dispone de un manejador de fichero se puede construir un bucle for para ir leyendo y contabilizando cada una de las líneas de un fichero.

```
In [ ]: manf = open('ejemplo.txt')
        contador = 0
        for linea in manf:
            contador = contador + 1
        print 'Líneas contabilizadas:', contador
```

# Ficheros

- Se usa el manejador del fichero como una secuencia en el bucle contando el número de líneas del fichero. Python se encarga de dividir los datos del fichero en líneas independientes, usando el carácter de salto de línea. Se lee cada línea hasta el salto de línea e incluye el propio salto de línea como último carácter en la variable línea para cada iteración del bucle for.

# Ficheros

- Cunado el fichero es relativamente pequeño comparado se puede leer el fichero completo en una cadena usando el método read sobre el manejador del fichero. Cuando el archivo se lee de esta manera, todos los caracteres incluyendo las líneas completas y los saltos de línea forman parte de la cadena.

```
In [ ]: manf = open('ejemplo.txt')
        ent = manf.read()
        print len(ent)
```

# Ficheros

- Cuando se buscan datos dentro de un fichero, se puede ir leyendo el archivo completo, ignorando determinadas líneas (utilizando en el bucle continue) y procesando únicamente aquellas que cumplen alguna condición particular. Y combinarlo con métodos de cadena.

```
In [ ]: manf = open('ejemplo.txt')
        for linea in manf:
                linea = linea.rstrip()
                if #condición de búsqueda :
                        continue
                 print linea
```

# Ficheros

- Se puede usar el método find para simular una búsqueda como la de un editor que texto que localiza aquellas líneas que contienen la cadena buscada en cualquier punto de las mismas. El método find comprueba la aparición de una cadena dentro de otra y devuelve la posición de la cadena o -1 si no la ha encontrado.

```
In [ ]: manf = open('ejemplo.txt')
        for linea in manf:
            linea = linea.rstrip()
            if linea.find('cadena buscada') == -1 :
                continue
            print linea
```

# Ficheros

- Si se quiere hacer un programa que permita abrir cualquier fichero, se le puede pedir al usuario que introduzca el nombre del fichero mediante raw_input y se guarda en una variable.

```
In [ ]: nombref = raw_input('Introduzca el nombre del fichero: ')
        manf = open(nombref)
        ...
```

# Ficheros

- A veces puede fallar una llamada a open, para lo cual se añade código de recuperación usando la estructura try/except:

```
In [ ]: nombref = raw_input('Introduzca el nombre del fichero: ')
        try:
            manf = open(nombref)
        except:
            print 'No se pudo abrir el fichero:', nombref
            exit()
        ....
```

- La función exit hace finalizar el programa.

# Ficheros

- Para escribir en un fichero, debes abrirlo usando el modo 'w' (de write) como segundo parámetro:

```
In [ ]: fsal = open('salida.txt', 'w')
```

- Si el fichero ya existe, abrirlo en modo escritura eliminará los datos antiguos y lo dejará completamente vacío, y si el fichero no existe, se creará uno nuevo.

# Ficheros

- El método write pone datos dentro de un archivo.

```
In [ ]: fsal = open('salida.txt', 'w')
        linea1 = "Esta es una nueva línea\n"
        fsal.write(linea1)
```

- Observar que cuando se llama a write otra vez se añaden los datos nuevos al final del archivo.

# Ficheros

- Cuando se escribe en un fichero hay que gestionar los finales de las líneas mientras se escribe en el fichero, insertando explícitamente el carácter de salto cuando se quiere terminar una línea. La sentencia print añade un salto de línea automáticamente, pero el método write no lo hace a menos que se lo especifiquemos.

# Ficheros

- Cuando se ha terminado de escribir un fichero se debe cerrar el fichero para asegurarse de que todo se guarda físicamente.

```
In [ ]: fsal.close()
```

- También se pueden cerrar los ficheros que se han abierto en modo lectura, pero no es necesario, ya que Python se asegura de que todos los ficheros queden cerrados cuando el programa termina.

# Ficheros

- Cuando se leen y escriben archivos se pueden tener problemas con los espacios en blanco, tabulaciones y saltos de línea pues normalmente son invisibles. En estos casos puede ser útil usar la función interna **repr** que toma cualquier objeto como argumento y devuelve una representación de cadena del objeto. En el caso de las cadenas, representa los caracteres en blanco con secuencias de barras invertidas.

# Listas

- Una lista es una secuencia de valores, cero o más, de cualquier tipo que reciben el nombre de elementos.
- El método más simple consiste en encerrar los elementos entre corchetes ([ y ]):

```
In [1]: [10, 20, 30, 40]
Out[1]: [10, 20, 30, 40]
```

# Listas

- En el ejemplo se crea una lista denominada **milista** que contiene 3 números. La asignación de valores a una lista no retorna nada, sin embargo si usamos el nombre de la lista, podemos ver el contenido de la variable.

```
In [23]: milista = [1,2,3]

In [24]: milista

Out[24]: [1, 2, 3]
```

# Listas

- La función range() crea una secuencia aritmética que es muy útil en los bucles.

```
In [25]: lista1 = range(0,10)
         lista2 = [0,1,2,3,4,5,6,7,8,9]

In [26]: print lista1
         print lista2

         [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
         [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# Listas

- Los elementos en una lista no tienen por qué ser todos del mismo tipo.

```
In [2]: ['spam', 2.0, 5, [10, 20]]
Out[2]: ['spam', 2.0, 5, [10, 20]]
```

- Una lista dentro de otra se dice que está anidada.
- Una lista que no contiene elementos recibe el nombre de lista vacía se puede crear una simplemente con unos corchetes vacíos, [].

# Listas

- Se pueden asignar listas de valores a variables:

```
In [3]: quesos = ['Cheddar', 'Edam', 'Gouda']
        numeros = [17, 123]
        vacia = []
        print quesos, numeros, vacia

['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

- Para acceder a los elementos de una lista se usa el operador corchete que contiene una expresión que especifica el índice (los índices comienzan por 0)

```
In [4]: print quesos[0]

Cheddar
```

# Listas

- Las listas son mutables porque pueden cambiar el orden de los elementos o reasignar un elemento dentro de la lista. Así cuando el operador corchete aparece en el lado izquierdo de una asignación éste identifica el elemento de la lista que será asignado.

```
In [5]: numeros = [17, 123]
        numeros[1] = 5
        print numeros

[17, 5]
```

# Listas

- En el siguiente ejemplo dada una lista se calcula los cuadrados de los valores de la lista.

```
In [33]: x = [1,3,7,2,8]
         y = []
         for val in x:
             y.append(val*val)
         y
Out[33]: [1, 9, 49, 4, 64]
```

- Observar:
  - Primero se crea una nueva lista vacia , asignando a una variable [].
  - Segundo se usa un bucle para iterar sobre la lista: for val in x:
  - Finalmente, se pueden añadir valores a una lista usando list.append(value) que añade un valor a la lista.

# Listas

- Otras operaciones que se pueden hacer:
  - Se pueden cambiar los valores existentes de una lista.

```
In [2]: y[1] = 100
        y
Out[2]: [1, 100, 49, 4, 64]
```

  - Soporta indexación con números negativos, que permite seleccionar por el final de la lista.

```
In [3]: y[-1]
Out[3]: 64
```

  - Se puede cambiar el primer y último valor de la lista.

```
In [4]: y[0],y[-1] = y[-1],y[0]
        y
Out[4]: [64, 100, 49, 4, 1]
```

# Listas

- La relación entre índices y elementos de una lista se denomina mapeo o direccionamiento.
- Los índices de una lista se caracterizan por:
  - Cualquier expresión entera puede ser utilizada como índice.
  - Si se intenta leer o escribir un elemento que no existe, se obtiene un IndexError.
  - Si un índice tiene un valor negativo, se cuenta hacia atrás desde el final de la lista.

# Listas

- El operador in también funciona con las listas.

```
In [6]: quesos = ['Cheddar', 'Edam', 'Gouda']
        'Edam' in quesos

Out[6]: True
```

# Listas

- Para recorrer los elementos de una lista y solo leer sus elementos se utiliza un bucle for como por ejemplo:

```
In [7]: for queso in quesos:
            print queso

Cheddar
Edam
Gouda
```

- Y si se quiere escribir o modificar lo elementos se necesitan los índices, para lo cual se usan las funciones range y len:

```
In [8]: for i in range(len(numeros)):
            numeros[i] = numeros[i] * 2
```

# Listas

- El bucle recorre la lista y actualiza cada elemento. len devuelve el número de elementos de la lista. range devuelve una lista de índices desde 0 hasta *n*−1, donde *n* es la longitud de la lista. Cada vez que atravesamos el bucle, i obtiene el índice del elemento siguiente. La sentencia de asignación en el cuerpo usa i para leer el valor antiguo del elemento y asignarle el valor nuevo.

# Listas

- Un bucle for aplicado a una lista vacía no ejecuta nunca el código contenido en su cuerpo.
- Una lista anidada sólo cuenta como un único elemento.

# Listas

- Algunos operadores de listas son:
  - El operador + concatena listas.

```
In [9]: a = [1, 2, 3]
        b = [4, 5, 6]
        c = a + b
        print c

        [1, 2, 3, 4, 5, 6]
```

  - El operador * repite una lista el n´umero especificado de veces.

```
In [10]: [1, 2, 3] * 3

Out[10]: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

# Listas

– El operador (slice) permite seleccionar secciones de una lista:

```
In [11]: t = ['a', 'b', 'c', 'd', 'e', 'f']
         t[1:3]

Out[11]: ['b', 'c']

In [12]: t[:4]

Out[12]: ['a', 'b', 'c', 'd']

In [13]: t[3:]

Out[13]: ['d', 'e', 'f']

In [14]: t[:]

Out[14]: ['a', 'b', 'c', 'd', 'e', 'f']
```

# Listas

- Un operador de slice en la parte izquierda de una asignación puede modificar múltiples elementos.

```
In [15]: t = ['a', 'b', 'c', 'd', 'e', 'f']
         t[1:3] = ['x', 'y']
         print t

['a', 'x', 'y', 'd', 'e', 'f']
```

# Listas

- Los principales métodos para manipular listas son:
  - append añade un nuevo elemento al final de una lista.

```
In [16]: t = ['a', 'b', 'c']
         t.append('d')
         print t

['a', 'b', 'c', 'd']
```

  - extend toma una lista como argumento y añade al final de la actual todos sus elementos.

```
In [17]: t1 = ['a', 'b', 'c']
         t2 = ['d', 'e']
         t1.extend(t2)
         print t1

['a', 'b', 'c', 'd', 'e']
```

# Listas

- sort ordena los elementos de una lista de menor a mayor.

```
In [18]: t = ['d', 'c', 'e', 'b', 'a']
         t.sort()
         print t

['a', 'b', 'c', 'd', 'e']
```

- Observar que la mayoría de los métodos de lista no devuelven nada; modifican la lista y devuelven None.

# Listas

- Hay varias formas de borrar elementos de una lista:
  - Si conoces el índice del elemento que se quiere eliminar se puede usar pop:

```
In [19]: t = ['a', 'b', 'c']
         x = t.pop(1)
         print t

         ['a', 'c']

In [20]: print x

         b
```

  - pop modifica la lista y devuelve el elemento que ha sido eliminado. Si no se proporciona un índice, borra y devuelve el útimo elemento.

# Listas

- Si no se necesita el valor eliminado se puede usar el operador del:

```
In [21]: t = ['a', 'b', 'c']
         del t[1]
         print t

['a', 'c']
```

- Si se conoce el elemento que se quiere eliminar (pero no su índice) se puedes usar remove.

```
In [22]: t = ['a', 'b', 'c']
         t.remove('b')
         print t

['a', 'c']
```

# Listas

- Para eliminar más de un elemento se puede usar del con un índice slice, que selecciona todos los elementos hasta (pero sin incluir) el segundo índice.

```
In [23]: t = ['a', 'b', 'c', 'd', 'e', 'f']
         del t[1:5]
         print t

['a', 'f']
```

# Listas

- Hay varias funciones internas que pueden utilizarse en las listas y que nos permiten buscar rápidamente a través de ellas:
  - La función sum() permite realizar la suma de una lista de números.
  - Las funciones máx() y min() proporcionan el elemento máximo/mínimo de una lista.
  - La función len() proporciona la longitud de una lista.

# Listas

```
In [24]: nums = [3, 41, 12, 9, 74, 15]
         print len(nums)

         6

In [25]: print max(nums)

         74

In [26]: print min(nums)

         3

In [27]: print sum(nums)

         154
```

# Listas

- Una cadena es una secuencia de caracteres y una lista es una secuencia de valores, pero una lista de caracteres no es lo mismo que una cadena. Para convertir desde una cadena a una lista de caracteres, se puede usar la función list que divide una cadena en letras individuales

```
In [28]: s = 'spam'
         t = list(s)
         print t

['s', 'p', 'a', 'm']
```

# Listas

- Si se quiere dividir una cadena en palabras, puedes usar el método split:

```
In [29]: s = 'suspirando por los fiordos'
         t = s.split()
         print t

['suspirando', 'por', 'los', 'fiordos']
```

- Una vez usado split se puede utilizar el operador índice (corchetes) para buscar una palabra concreta en la lista.

```
In [30]: print t[2]

los
```

# Listas

- Se puede llamar a split con un argumento opcional llamado delimitador, que especifica qué caracteres se deben usar como delimitadores de palabras.

```
In [31]: s = 'spam-spam-spam'
         delimitador = '-'
         s.split(delimitador)

Out[31]: ['spam', 'spam', 'spam']
```

# Listas

- join es la inversa de Split y toma una lista de cadenas y concatena sus elementos. Al ser un método de cadena debe invocarse sobre el delimitador y pasarle la lista como un parámetro.

```
In [32]: t = ['suspirando', 'por', 'los', 'fiordos']
         delimitador = ' '
         delimitador.join(t)

Out[32]: 'suspirando por los fiordos'
```

- En caso de que el delimitador sea el carácter espacio, entonces join coloca un espacio entre las palabras. Para concatenar cadenas sin espacios, puedes usar la cadena vacía, '', como delimitador.

# Listas

- En Python las listas son objetos de manera que dos listas son equivalentes, si tienen los mismos elementos, pero no son idénticas, porque no son el mismo objeto. En este sentido si dos objetos son idénticos, también son equivalentes, pero si son equivalentes no necesariamente son idénticos.
- Para comprobar si dos variables se refieren al mismo objeto se puede usar el operador is.

```
In [33]: a = [1, 2, 3]
         b = [1, 2, 3]
         a is b

Out[33]: False
```

# Listas

- Si a se refiere a un objeto y se asigna b = a, entonces ambas variables se refieren al mismo objeto:

```
In [34]: a = [1, 2, 3]
         b = a
         b is a
Out[34]: True
```

- La asociación de una variable con un objeto recibe el nombre de referencia. En este ejemplo, hay dos referencias para el mismo objeto. Un objeto con más de una referencia tiene más de un nombre, de modo que se dice que el objeto tiene uno o varios alias.

# Listas

- Si un objeto con alias es mutable, los cambios que se hagan en uno de los alias afectarán al otro.

```
In [35]: a = [1, 2, 3]
         b = a
         b[0] = 17
         print a

         [17, 2, 3]
```

# Listas

- Cuando se pasa una lista a una función, la función recibe una referencia de esa lista. Si la función modifica un parámetro de la lista, el código que la ha llamado también se verá afectado por el cambio.

```
In [36]: def borra_primer(t):
             del t[0]
         letras = ['a', 'b', 'c']
         borra_primer(letras)
         print letras

['b', 'c']
```

# Listas

- En las operaciones que se realizan sobre las listas hay operaciones que modifican listas y otras que crean listas nuevas. Por ejemplo, el método append modifica una lista, pero el operador + crea una lista nueva.

```
In [37]: t1 = [1, 2]
         t2 = t1.append(3)
         print t1

         [1, 2, 3]

In [38]: print t2

         None

In [39]: t3 = t1 + [3]
         print t3

         [1, 2, 3, 3]

In [40]: t2 is t3

Out[40]: False
```

# Listas

- Observar:
  - La mayoría de los métodos de las listas modifican el argumento y devuelven None frente a los métodos de cadena que devuelven una cadena nueva y dejan la original inalterada.
  - Es mejor hacer copias para evitar los alias.

# Listas

- En el siguiente ejemplo, se toma una simple lista, y se retornan dos listas. La primera lista que contiene los valores más pequeños que un cierto número y otra que contiene los valores que son mayores o igual que ese número.
  - listainicial = [1,5,6,8,1,12,5,2,7,8,9]
  - numero = 5
  - valoresmenores=[]
  - valoresgrandes=[]

# Listas

In [1]:
```
# Solución

listainicial = [1,5,6,8,1,12,5,2,7,8,9]
cutoff = 5
valoresmenores=[]
valoresgrandes=[]

for v in listainicial:
    if (v < cutoff):
        valoresmenores.append(v)
    else:
        valoresgrandes.append(v)

valoresgrandes.sort(reverse=True)
valoresmenores.sort(reverse=True)

print valoresmenores
print valoresgrandes
```

```
[2, 1, 1]
[12, 9, 8, 8, 7, 6, 5, 5]
```

# Diccionarios

- Un diccionario es una asignación entre un conjunto de índices (a los cuales se les llama claves) y un conjunto de valores. Cada clave apunta a un valor.
- Los índices pueden ser de (casi) cualquier tipo.

# Diccionarios

- La función dict() crea un diccionario nuevo sin elementos.

```
In [2]: ejemplo = dict()
        print ejemplo
{}
```

- Las llaves {}, representan un diccionario vacío. Para añadir elementos al diccionario se pueden usar corchetes.

```
In [4]: ejemplo['primero'] = 'uno'
        print ejemplo
{'primero': 'uno'}
```

# Diccionarios

- Otra forma de crear un diccionario es mediante una secuencia de pares clave-valor separados por comas y encerrados entre llaves.

```
In [5]: ejemplo2 = {'primero': 'uno', 'segundo': 'dos', 'tercero': 'tres'}
        print ejemplo2

{'tercero': 'tres', 'segundo': 'dos', 'primero': 'uno'}
```

- El orden de los elementos en un diccionario es impredecible, pero eso no es importante dado que se usan las claves para buscar los valores correspondientes. En este sentido i la clave especificada no está en el diccionario se obtiene una excepción.

# Diccionarios

- Algunos métodos:
  - La función **len** devuelve el número de parejas clave-valor.

```
In [6]: len(ejemplo2)
Out[6]: 3
```

  - El operador **in** dice si algo aparece como c*lave* en el diccionario.

```
In [7]: 'primero' in ejemplo2
Out[7]: True
```

# Diccionarios

- Para ver si algo aparece como valor en un diccionario, se puede usar el método **values**, que devuelve los valores como una lista, y después usar el operador in sobre esa lista.

```
In [8]: valores = ejemplo2.values()
        'uno' in valores

Out[8]: True
```

# Diccionarios

- El método **get** toma una clave y un valor por defecto. Si la clave aparece en el diccionario get devuelve el valor correspondiente. En caso contrario devuelve el valor por defecto.

```
In [10]: contadores = { 'naranjas' : 1 , 'limones' : 42, 'peras': 100}
         print contadores.get('uvas', 0)

         0
```

# Diccionarios

- El método keys crea una lista con las claves de un diccionario.

```
In [12]: contadores.keys()

Out[12]: ['naranjas', 'limones', 'peras']
```

# Diccionarios

- Observar que si se utiliza un diccionario como secuencia en una sentencia for, entonces se recorre todas las claves del diccionario.

```
In [11]: contadores = { 'naranjas' : 1 , 'limones' : 42, 'peras': 100}
         for clave in contadores:
             print clave, contadores[clave]

naranjas 1
limones 42
peras 100
```

# Tuplas

- Una tupla es una secuencia de valores que pueden ser de cualquier tipo indexada por enteros.

- Las tuplas son inmutables, comparables y dispersables (hashables), de modo que las listas de tuplas se pueden ordenar y es posible usar tuplas como valores para las claves en los diccionarios.

# Tuplas

- Sintácticamente, una tupla es una lista de valores separados por comas y encerradas entre paréntesis.

```
In [13]: t = ('a', 'b', 'c', 'd', 'e')
```

- Para crear una tupla con un ´unico elemento, es necesario incluir una coma al final, en caso contrario es tratado como una expresión con una cadena dentro de un paréntesis, que evalúa como de tipo “string”.

```
In [15]: t1 = ('a',)
         type(t1)

Out[15]: tuple

In [16]: t2 = ('a')
         type(t2)

Out[16]: str
```

# Tuplas

- Otra forma de construir una tupla es usar la función interna tuple que crea una tupla vacía si se invoca sin argumentos, y si se le proporciona como argumento una secuencia (cadena, lista o tupla) genera una tupla con los elementos de la secuencia.

```
In [17]: t = tuple()
         print t

         ()

In [19]: t = tuple('altramuces')
         print t

         ('a', 'l', 't', 'r', 'a', 'm', 'u', 'c', 'e', 's')
```

# Tuplas

- La mayoría de los operadores de listas funcionan también con tuplas:
  - El operador corchete indexa un elemento.

```
In [20]: t = ('a', 'b', 'c', 'd', 'e')
         print t[0]

a
```

  - El operador slice selecciona un rango de elementos.

```
In [21]: print t[1:3]

('b', 'c')
```

# Tuplas

- No se pueden modificar los elementos de una tupla, pero se puede reemplazar una tupla con otra.

```
In [22]: t = ('A',) + t[1:]
         print t

('A', 'b', 'c', 'd', 'e')
```

# Tuplas

- Los operadores de comparación funcionan también con las tuplas de forma que comienza comparando el primer elemento de cada secuencia. Si es igual en ambas, pasa al siguiente elemento, y así sucesivamente, hasta que encuentra uno que es diferente. A partir de ese momento, los elementos siguientes ya no son tenidos en cuenta.

```
In [23]: (0, 1, 2) < (0, 3, 4)

Out[23]: True
```

# Tuplas

- La función sort funciona del mismo modo. Ordena por el primer elemento, pero en caso de que haya dos iguales, usa el segundo, y así sucesivamente.

# Tuplas

- En Python es posible tener una tupla en el lado izquierdo de una sentencia de asignación, lo que permite asignar varias variables el mismo tiempo cuando tenemos una secuencia en el lado izquierdo.

```
In [24]: m = [ 'p´asalo', 'bien' ]
         x, y = m
         x

Out[24]: 'p\xc2\xb4asalo'

In [25]: y

Out[25]: 'bien'
```

- La cantidad de variables en el lado izquierdo y la cantidad de valores en el derecho debe ser la misma, y en general el lado derecho puede ser cualquier tipo de secuencia (cadena,lista o tupla).

# Tuplas

- Los diccionarios tienen un método llamado items que devuelve una lista de tuplas, cada una de las cuales es una pareja clave-valor sin ningún orden definido.

```
In [26]: d = {'a':10, 'b':1, 'c':22}
         t = d.items()
         print t

[('a', 10), ('c', 22), ('b', 1)]
```

# Tuplas

- La combinación de items, asignación en tupla y for permite recorrer las claves y valores de un diccionario en un único bucle:

```
for clave, valor in d.items():
    print valor, clave
```

# Tuplas

- Este bucle tiene dos variables de iteración, ya que items devuelve una lista de tuplas y clave, valor es una asignación en tupla, que itera sucesivamente a través de cada una de las parejas clave-valor del diccionario. Para cada iteración a través del bucle, tanto clave como valor van pasando a la siguiente pareja clave-valor del diccionario (todavía en orden de dispersión).

# Tuplas

- Dado que las tuplas son dispersables (hashables) y las listas no, si se quiere crear una clave compuesta para usar en un diccionario, se debe usar una tupla como clave.

```
In [ ]: directorio[apellido,nombre] = numero
```

- La expresión dentro de los corchetes es una tupla, y se podría usar asignaciones mediante tuplas en un bucle for para recorrer el diccionario.

```
In [ ]: for apellido, nombre in directorio:
                    print nombre, apellido, directorio[apellido, nombre]
```

# Tuplas

- Dado que las tuplas son inmutables, no proporcionan métodos sort y reverse, que modifican listas ya existentes, pero proporciona las funciones integradas sorted y reversed que toman una secuencia como parámetro y devuelven una secuencia nueva con los mismos elementos en un orden diferente.

# Tuplas

- Observaciones sobre el uso de secuencias(listas, tuplas y cadenas):
  - Las cadenas están más limitadas que las demás secuencias, porque los elementos deben ser caracteres, y además son inmutables. Es por ello que si se necesita la capacidad de cambiar los caracteres en una cadena (en vez de crear una nueva) es más adecuado elegir una lista de caracteres.
  - Las listas se usan con más frecuencia que las tuplas porque son mutables.

# Tuplas

- En una sentencia return es más simple crear una tupla que una lista.
- Si se quiere usar una secuencia como una clave en un diccionario, debe usarse un tipo inmutable como una tupla o una cadena.
- Si se está pasando una secuencia como argumento de una función, el uso de tuplas evita el problema de la creación de alias.

# Clases

- Las clases se definen mediante la palabra clave class seguida del nombre de la clase, dos puntos (:) y a continuación, indentado, el cuerpo de la clase. Si la primera línea del cuerpo se trata de una cadena de texto, esta será la cadena de documentación de la clase o docstring.

# Clases

```
In [6]: class Coche:
          '''Abstraccion de los objetos coche.'''
          def __init__(self, gasolina):
              self.gasolina = gasolina
              print "Tenemos", gasolina, "litros"
          def arrancar(self):
              if self.gasolina > 0:
                 print "Arranca"
              else:
                 print "No arranca"
          def conducir(self):
              if self.gasolina > 0:
                 self.gasolina -= 1
                 print "Quedan", self.gasolina, "litros"
              else:
                 print "No se mueve"
```

# Clases

- El método __init__ con una doble barra baja al principio y final del nombre, se ejecuta justo después de crear un nuevo objeto a partir de la clase y sirve para realizar cualquier proceso de inicialización.
- El primer parámetro de __init__ y del resto de métodos de la clase es siempre self permite acceder a los atributos y métodos del objeto diferenciando, por ejemplo, una variable local mi_var de un atributo del objeto self.mi_var.

# Clases

- En el método __init__ de la clase Coche se utiliza self para asignar al atributo gasolina del objeto (self.gasolina) el valor especificado para el parámetro gasolina. El parámetro gasolina se destruye al final de la función, mientras que el atributo gasolina se conserva (y puede ser accedido) mientras el objeto exista.

# Clases

- Para crear un objeto se escribe el nombre de la clase seguido de cualquier parámetro que sea necesario entre paréntesis. Estos parámetros son los que se pasarán al método __init__.

```
In [7]: mi_coche = Coche(3)

Tenemos 3 litros
```

- Observar que aunque __init__ tiene 2 parámetros (self y gasolina) solo se pasa un valor debido a que el primer argumento (la referencia al objeto que se crea) se pasa automáticamente.

# Clases

- Una vez creado el objeto se puede acceder a los atributos y métodos mediante la sintaxis objeto.atributo y objeto.metodo():

```
In [8]: print mi_coche.gasolina
3
In [9]: mi_coche.arrancar()
Arranca
In [10]: mi_coche.conducir()
Quedan 2 litros
In [11]: mi_coche.conducir()
Quedan 1 litros
In [12]: mi_coche.conducir()
Quedan 0 litros
In [13]: mi_coche.conducir()
No se mueve
In [14]: print mi_coche.gasolina
0
```

# Clases

- Para indicar que una clase hereda de otra se coloca el nombre de la clase de la que se hereda entre paréntesis después del nombre de la clase.

```
In [17]: class Instrumento:
           def __init__(self, precio):
                 self.precio = precio
           def tocar(self):
                 print "Estamos tocando musica"
           def romper(self):
                 print "Eso lo pagas tu"
                 print "Son", self.precio, "$$$"
         class Bateria(Instrumento):
             pass
         class Guitarra(Instrumento):
             pass
```

# Clases

- Se ha creado una clase Instrumento con las atributos y métodos comunes a todos los instrumentos y las clases Bateria y Guitarra heredan de Instrumento el método tocar() y el método romper(), y se inicializan pasando un parámetro precio.

# Clases

- Si se quiere especificar un nuevo parámetro tipo_cuerda cuando se crea un objeto Guitarra bastaría con escribir un nuevo método __init__ para la clase Guitarra que se ejecutaría en lugar del __init__ de Instrumento. Esto es lo que se conoce como sobreescribir métodos.

# Clases

- Puede ocurrir que se necesite sobreescribir un método de la clase padre, pero que en ese método se quiera ejecutar el método de la clase padre porque el nuevo método no necesite más que ejecutar un par de nuevas instrucciones extra. En ese caso se usaría la sintaxis SuperClase.metodo(self, args) para llamar al método de igual nombre de la clase padre. Por ejemplo, para llamar al método __init__ de Instrumento desde Guitarra se usaría Instrumento.__init__(self, precio). En este caso si es necesario especificar el parámetro self.

# Clases

- En Python se permite la herencia múltiple, es decir, una clase puede heredar de varias clases a la vez para lo cual basta enumerar las clases de las que se hereda separándolas por comas.

- En el caso de que alguna de las clases padre tuvieran métodos con el mismo nombre y número de parámetros las clases sobreescribirían la implementación de los métodos de las clases más a su derecha en la definición.

# Clases

In [20]:
```
class Terrestre:
    def desplazar(self):
        print "El animal anda"
class Acuatico:
    def desplazar(self):
        print "El animal nada"
class Cocodrilo(Terrestre, Acuatico):
    pass
c = Cocodrilo()
c.desplazar()
```

```
El animal anda
```

# Clases

- La clase Cocodrilo hereda de la clase Terrestre y de la clase Acuatico. Como ambas clases tienen métodos con el mismo nombre y número de parámetros y dado que Terrestre se encuentra más a la izquierda, sería la definición de desplazar de esta clase la que prevalece, y por lo tanto al llamar al método desplazar de un objeto de tipo Cocodrilo se imprime "El animal anda".

# Clases

- La palabra polimorfismo se refiere a la habilidad de objetos de distintas clases de responder al mismo mensaje. Python, al ser de tipado dinámico, no impone restricciones a los tipos que se le pueden pasar a una función, por lo que el polimorfismo en Python no es de gran importancia.

# Clases

- La sobrecarga de métodos (capacidad del lenguaje de determinar qué método ejecutar de entre varios métodos con igual nombre según el tipo o número de los parámetros que se le pasa) no existe en Python pues el último método sobrescribiría la implementación de los anteriores. Sin embargo se puede conseguir un comportamiento similar como por ejemplo recurriendo a funciones con valores por defecto para los parámetros.

# Clases

- La encapsulación se refiere a impedir el acceso a determinados métodos y atributos de los objetos estableciendo así qué puede utilizarse desde fuera de la clase. En Python el acceso a una variable o función viene determinado por su nombre: si el nombre comienza con dos guiones bajos (y no termina también con dos guiones bajos) se trata de una variable o función privada, en caso contrario es pública.

# Clases

```
class Ejemplo:
    def publico(self):
        print "Publico"
    def __privado(self):
        print "Privado"
ej = Ejemplo()
ej.publico()
ej.__privado()
```

```
Publico
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-21-a36193b80ac3> in <module>()
      6 ej = Ejemplo()
      7 ej.publico()
----> 8 ej.__privado()

AttributeError: Ejemplo instance has no attribute '__privado'
```

# Clases

- En el ejemplo sólo se imprimirá la cadena correspondiente al método publico(), mientras que al intentar llamar al método __privado() se lanza una excepción quejándose de que no existe. Este mecanismo se basa en que los nombres que comienzan con un doble guión bajo se renombran para incluir el nombre de la clase (*name mangling*). Esto implica que el método o atributo no es realmente privado, y se podría acceder con ej._Ejemplo__privado()

# Clases

- En ocasiones puede suceder que se quiera permitir un acceso controlado a algún atributo de un objeto. Para ello se escriben métodos para este cometido y cuyos nombres son getVariable y setVariable.

```
In [23]: class Fecha():
             def __init__(self):
                 self.__dia = 1
             def getDia(self):
                 return self.__dia
             def setDia(self, dia):
                 if dia > 0 and dia < 31:
                     self.__dia = dia
                 else:
                     print "Error"
         mi_fecha = Fecha()
         mi_fecha.setDia(33)

         Error
```

# Clases

- Este mecanismo se puede simplificar mediante propiedades, que abstraen al usuario del hecho de que se está utilizando métodos

```
In [24]: class Fecha(object):
             def __init__(self):
                 self.__dia = 1
             def getDia(self):
                 return self.__dia
             def setDia(self, dia):
                 if dia > 0 and dia < 31:
                     self.__dia = dia
                 else:
                     print "Error"
             dia = property(getDia, setDia)
         mi_fecha = Fecha()
         mi_fecha.dia = 33

         Error
```

# Clases

- En el ejemplo anterior la clase Fecha deriva de object dado que se trata de una clase enriquecida que añade nuevas funcionalidades con respecto a las clases clásicas tales como descriptores, métodos estáticos,…
- Para que una clase sea enriquecida es necesario que extienda una clase de enriquecida. En el caso de que no sea necesario heredar el comportamiento o el estado de ninguna clase se puede heredar de object, que es un objeto vacio que sirve como base para todas las clases enriquecidas.

# Clases

- La diferencia principal entre las clases antiguas y las enriquecidas consiste en que cuando se crea una nueva clase antes no se definía realmente un nuevo tipo, sino que todos los objetos creados a partir de clases eran de tipo instance.

# Clases

- Además del método __init__, existen otros métodos con significados especiales, cuyos nombres siempre comienzan y terminan con dos guiones bajos:
  - __init__(self, args): Método llamado después de crear el objeto para realizar tareas de inicialización.
  - __del__(self): Método llamado cuando el objeto va a ser borrado. También llamado destructor, se utiliza para realizar tareas de limpieza.

# Clases

- __str__(self): Método llamado para crear una cadena de texto que represente a un objeto. Se utiliza cuando se usa print para mostrar un objeto o cuando se usa la función str(obj) para crear una cadena a partir de un objeto.

# Clases

- __new__(cls, args): Método exclusivo de las clases enriquecidas que se ejecuta antes que __init__ y que se encarga de construir y devolver el objeto en sí. Se trata de un método estático, es decir, que existe con independencia de las instancias de la clase: es un método de clase, no de objeto, y por lo tanto el primer parámetro no es self, sino la propia clase: cls.

# Clases

- __cmp__(self, otro): Método llamado cuando se utilizan los operadores de comparación para comprobar si un objeto es menor, mayor o igual al objeto pasado como parámetro. Debe devolver un número negativo si el objeto es menor, cero si son iguales, y un número positivo si el objeto es mayor. Si este método no está definido y se intenta comparar el objeto mediante los operadores <, <=, > o >= se lanzará una excepción. Si se utilizan los operadores == o != para comprobar si dos objetos son iguales, se comprueba si son el mismo objeto (si tienen el mismo id).

# Clases

- __len__(self): Método llamado para comprobar la longitud del objeto. Se utiliza, por ejemplo, cuando se llama a la función len(obj) sobre un objeto.

# Expresiones regulares

- La librería de expresiones regulares se denomina re y debe ser importada en el programa antes de poder utilizarlas. El uso más simple es la función search() que permite buscar una cadena en un texto.

# Expresiones regulares

- En el siguiente programa, se abre el fichero, se recorre cada línea, y mediante la función search() se imprimen solamente aquellas líneas que contienen la cadena “Casa”.

```
import re
manf = open('texto.txt')
for linea in manf:
    linea = linea.rstrip()
    if re.search('Casa', linea) :
        print linea
```

# Expresiones regulares

- Observar que esta misma operación se podría haber realizado con el método linea.find() de la librería de cadenas.

# Expresiones regulares

- Para localizar patrones más complejos se utilizan caracteres especiales que se añaden a la cadena de búsqueda. Por ejemplo, el carácter ^ indica “el comienzo” de una línea.

```
import re
manf = open('texto.txt')
for linea in manf:
    linea = linea.rstrip()
    if re.search('^Casa', linea) :
        print linea
```

# Expresiones regulares

- El programa ahora solo recupera las líneas que comiencen con la cadena “Casa”. Sin embargo, esta misma operación se podría haber realizado con el método startswith() de la librería de cadenas.

# Expresiones regulares

- Hay varios caracteres especiales que permiten construir expresiones regulares:
  - El punto equivale a cualquier carácter.

```
import re
manf = open('texto.txt')
for linea in manf:
    linea = linea.rstrip()
    if re.search('^F..m:', linea) :
        print linea
```

# Expresiones regulares

- Otros caracteres indican que un carácter puede repetirse cualquier número de veces, usando “*” (cero-o-más caracteres ) o “+”(uno-o-más caracteres) en la expresión regular. El “*” o “+” se aplica al carácter que queda inmediatamentea la izquierda del más o del asterisco.

```
import re
manf = open('texto.txt')
for linea in manf:
    linea = linea.rstrip()
    if re.search('^From:.+@', linea) :
        print linea
```

# Expresiones regulares

- En este ejemplo la cadena “^From:.+@” encuentra todas las líneas que comienzan con “From:”, seguido por uno o más caracteres (“.+”), seguidos de un símbolo arroba.

# Expresiones regulares

- Si se quiere extraer datos desde una cadena se puede usar el método findall() para obtener todas las subcadenas que coinciden con una expresión regular. El resultado devuelto por el método es una lista con los resultados.

# Expresiones regulares

- Considerar el problema de extraer todas las direcciones de correo electrónico de un texto. En este caso se puede usar el método findall() para buscar en el texto usando una expresión regular como \S+@\S+, donde \S equivale a cualquier carácter distinto de un espacio en blanco.

# Expresiones regulares

- La expresión busca subcadenas que tengan al menos un carácter que no sea un espacio en blanco, seguido por un signo arroba, seguido por al menos un carácter más que tampoco sea un espacio en blanco

```
import re
manf = open('texto.txt')
for linea in manf:
    linea = linea.rstrip()
    x = re.findall('\S+@\S+', linea)
    if len(x) > 0 :
        print x
```

# Expresiones regulares

- El programa va leyendo cada línea y extrae todas las subcadenas que coinciden con la expresión regular. Como findall() devuelve una lista, se comprueba si el número de elementos en la lista es mayor que cero con el objetivo de mostrar sólo aquellas líneas en las que se ha encontrado al menos una subcadena que parece una dirección de e-mail.

# Expresiones regulares

- En el ejemplo anterior se puede restringir aún más indicando que  la cadena que comience y termine con una letra o un número. Para ello se usan corchetes que indican un conjunto de varios caracteres aceptables en las coincidencias: [a-zA-Z0-9]\S*@\S*[a-zA-Z]

# Expresiones regulares

- La expresión indica que se están buscando subcadenas que comiencen con una única letra (minúscula o mayúscula), o un número “[a-zA-Z0-9]”, seguido por cero o más caracteres no-en-blanco (“\S*”), seguidos por un símbolo arroba, seguidos por cero o más caracteres no-en-blanco (“\S*”), seguidos por una letra mayúscula o minúscula. Observar que se ha cambiado de “+” a “*” para indicar cero o más caracteres no-blancos, dado que “[a-zA-Z0-9]” ya es un carácter no-blanco.

# Expresiones regulares

- El código queda como:

```
import re
manf = open('texto.txt')
for linea in manf:
    linea = linea.rstrip()
    x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', linea)
    if len(x) > 0 :
        print x
```

# Expresiones regulares

- Considerar que se tiene un texto del que se quieren recuperar los números de aquellas líneas que son de la siguiente forma:

  X-Codigo-B:9.9000

  X-Codigo-C:8.9944

# Expresiones regulares

- Para ello se puede construir una expresión regular para usar con findall() de la siguiente forma: ^X\S*: ([0-9.]+) que representa todas las cadenas que comienzan por X seguidas por cero o más caracteres (".*"), seguidas por dos-puntos (":") y luego un espacio. Después del espacio busca uno o más caracteres que sean o bien dígitos (0-9) o puntos "[0-9.]+"(dentro de los corchetes, el punto coincide con un punto real y no como un comodín). Los paréntesis indican que a pesar de que se desea que la expresión completa coincida, sólo se está interesado en extraer una cierta porción de la subcadena.

# Expresiones regulares

```
import re
manf = open('mbox-short.txt')
for linea in manf:
    linea = linea.rstrip()
    x = re.findall('^X\S*: ([0-9.]+)', linea)
    if len(x) > 0 :
        print x
```

# Expresiones regulares

- Por último considerar el problema de dado un texto procedente de un email donde la cabecera del mismo contiene líneas de la forma:

  FROM pepito.palotes@ucm.es Sat Jun 5 09:10:23 2015
- Se quiere extraer la hora del día de cada línea.

# Expresiones regulares

- Una primera solución consistiría en utilizar split(). Se divide la línea en palabras y luego se coge la quinta palabra y se divide de nuevo usando el carácter dos-puntos para extraer los dos caracteres.

```
datos="FROM pepito.palotes@ucm.es Sat Jun 5 09:10:23 2015"
palabras=datos.split()
palabra=palabras[5]
caracteres=palabra.split(':')
print caracteres[0]
```

# Expresiones regulares

- Una segunda solución consiste en utilizar una expresión regular con findall de la forma: ^From .* ([0-9][0-9]): , que representa líneas que comiencen con “From ”,seguidas por cualquier cantidad de caracteres (“.*”), seguidos por un espacio, seguidos por dos dígitos “[0-9][0-9]”, seguidos por un carácter dos puntos. Para extraer sólo la hora se usan los paréntesis alrededor de los dos dígitos.

# Expresiones regulares

- Por tanto el código queda como:

```
import re
datos='From pepito.palotes@ucm.es Sat Jun 5 09:10:23 2015'
datos=datos.strip()
x = re.findall('^From .* ([0-9][0-9]):', datos)
if len(x) > 0 :
    print x
```

# Expresiones regulares

- A veces interesa que los caracteres especiales sean un carácter normal, en estos casos se pone delante de ese carácter una barra invertida. En el siguiente ejemplo se busca en la cadena dada el coste en dólares.

```
import re
datos = 'El boligrafo cuesta 10.00$'
salida = re.findall('[0-9.]+\$',datos)
print salida
```

# Expresiones regulares

- Los principales caracteres especiales y secuencias que se usan en las expresiones regulares son:
  - ^ Coincide con el principio de una línea.
  - $ Coincide con el final de una línea.
  - . Coincide con cualquier carácter.
  - \s Coincide con un carácter espacio en blanco.
  - \S Coincide con cualquier carácter que no sea un espacio en blanco.

# Expresiones regulares

- * Se aplica al carácter que le precede e indica que la búsqueda debe coincidir cero o más veces con ´el.
- *? Se aplica al carácter que le precede e indica que la búsqueda debe coincidir cero o más veces con él ,buscando la cadena coincidente más pequeña posible.
- + Se aplica al carácter que le precede e indica que la búsqueda debe coincidir una o más veces con él.
- +? Se aplica al carácter que le precede e indica que la búsqueda debe coincidir una o más veces con él, buscando la cadena coincidente más pequeña posible.

# Expresiones regulares

- [aeiou] Coincide con un único carácter siempre que ese carácter esté en el conjunto especificado. En este ejemplo, deberían coincidir “a”, “e”, “i”, “o”, o “u”, pero no los demás caracteres.
- [a-z0-9] Se pueden especificar rangos de caracteres usando el guion. Este ejemplo indica un único carácter que puede ser una letra minúscula o un digito.

# Expresiones regulares

- [^A-Za-z] Cuando el primer carácter en la notación del conjunto es un símbolo de intercalación, se invierte la lógica. En este ejemplo, la expresión equivale a un único carácter que sea cualquier cosa excepto una letra mayúscula o minúscula.
- ( ) Cuando se añaden paréntesis a una expresión regular, éstos son ignorados durante la búsqueda, pero permiten extraer un subconjunto particular de la cadena localizada en vez de la cadena completa, cuando se usa findall().

# Expresiones regulares

- \b Coincide con la cadena vacía, pero solo al principio o al final de una palabra.
- \B Coincide con la cadena vacía, pero no al principio o al final de una palabra.
- \d Coincide con cualquier digito decimal, es equivalente al conjunto [0-9].
- \D Coincide con cualquier carácter que no sea un digito; equivale al conjunto [^0-9].

# Ayuda en Python

- Python tiene cierta documentación sencilla que puede verse en el interprete de Python en modo interactivo. Se puede acceder al sistema de ayuda interactivo usando help().

```
>>> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

# Ayuda en Python

- Si se conoce qué módulo se quiere usar, se puede utilizar el comando dir() para localizar los métodos del módulo:

```
>>> import re
>>> dir(re)
['DEBUG', 'DOTALL', 'I', 'IGNORECASE', 'L', 'LOCALE', 'M', 'MULTILINE', 'S', 'Scanner', 'T', 'TEMPLATE', 'U', 'UNICODE', 'VERBOSE', 'X', '_MAXCACHE', '_all_', '_bui
ltins_', '_doc_', '_file_', '_name_', '_package_', '_version_', '_alphanum', '_cache', '_cache_repl', '_compile', '_compile_repl', '_expand', '_locale', '_p
attern_type', '_pickle', '_subx', 'compile', 'copy_reg', 'error', 'escape', 'findall', 'finditer', 'match', 'purge', 'search', 'split', 'sre_compile', 'sre_parse', 'su
b', 'subn', 'sys', 'template']
```

# Ayuda en Python

- También se puede obtener un poco de documentación acerca de un método particular usando el comando help.

```
>>> help(re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
```