

# Colecciones, Iterables

Listas, Tuplas, Diccionarios, Cadenas, Conjuntos

Antonio Espín Herranz

# Listas

- La lista es un tipo de colección ordenada. Sería equivalente a lo que en otros lenguajes se conoce por arrays, o vectores.
- Los índices son numéricos y empiezan en 0.
- Las listas pueden contener cualquier tipo de dato: números, cadenas, booleanos, ... y también listas.
- Las listas se representan por los elementos entre corchetes.  
L = [22, True, "una lista", [1, 2]]

# Acceso a los elementos de las Listas

- Los elementos de una lista son accesibles por la posición que ocupan y empiezan en la posición 0.
  - `L = [11, False]`
  - `mi_var = L[0]` # mi\_var vale 11
  - `L2 = ["una lista", [1, 2]]`
  - `L2[1][0]` # Accede al elemento 1
  - `print(type(L))` → **list**
- También se pueden modificar los elementos de la lista:
  - `L2[0] = 99`

# índices negativos

- Podemos acceder a los elementos del final de la lista utilizando un número negativo dentro de los corchetes.
- Con [-1] accederíamos al último elemento de la lista, con [-2] al penúltimo, con [-3], al antepenúltimo, y así sucesivamente.

# Slicing

- Si en lugar de un número escribimos dos números inicio y fin separados por dos puntos **[inicio:fin]** Python interpretará que queremos una lista que vaya desde la posición inicio a la posición fin, sin incluir este último.
- Si escribimos tres números **[inicio:fin:salto]** en lugar de dos, el tercero se utiliza para determinar cada cuantas posiciones añadir un elemento a la lista.

```
l = [99, True, "una lista", [1, 2]]
```

```
mi_var = l[0:2] # mi_var vale [99, True]
```

```
mi_var = l[0:4:2] # mi_var vale [99, "una lista"]
```

# Slicing

- No es necesario incluir el último elemento:

```
l = [99, True, "una lista"]
```

```
mi_var = l[1:] # mi_var vale [True, "una lista"]
```

**[1:]** → del uno hasta el final.

```
mi_var = l[:2] # mi_var vale [99, True]
```

```
mi_var = l[:] # mi_var vale [99, True, "una lista"]
```

```
mi_var = l[::2] # mi_var vale [99, "una lista"]
```

# El ::2 representa un salto de dos en dos.

# Slicing

- También se puede utilizar el operador `[]` para modificar el contenido de la lista:

```
l = [99, True, "una lista", [1, 2]]
```

```
l[0:2] = [0, 1] # l vale [0, 1, "una lista", [1, 2]]
```

- Se puede **modificar el tamaño de la lista** si la lista de la parte derecha de la asignación tiene un tamaño menor o mayor que el de la selección de la parte izquierda de la asignación:

```
l[0:2] = [False] # l vale [False, "una lista", [1, 2]]
```

# Tuplas

- Las tuplas son similares a las listas, a excepción de la forma de definirla, para lo que se utilizan paréntesis en lugar de corchetes.

```
t = (1, 2, True, "python")
```

- En realidad el **constructor de la tupla es la coma**, no el paréntesis, pero el intérprete muestra los paréntesis, y nosotros deberíamos utilizarlos, por claridad.

```
>>> t = 1, 2, 3
```

```
– >>> type(t)
```

```
– type "tuple"
```



# tuplas

- **Ojo**, para tuplas de un solo elemento:
  - Hay que tener en cuenta que es necesario añadir una coma para tuplas de un solo elemento, para diferenciarlo de un elemento entre paréntesis:

```
>>> t = (1)
>>> type(t)
type "int"
```

```
>>> t = (1,)
>>> type(t)
type "tuple"
```

# tuplas

- Acceso a los elementos de las tuplas:
  - `mi_var = t[0]` # `mi_var` es 1
  - `mi_var = t[0:2]` # `mi_var` es (1, 2)
- Podemos utilizar el operador `[]` debido a que las tuplas, al igual que las listas, forman parte de un tipo de objetos llamados secuencias.
- **Las cadenas de texto también son secuencias**, por lo que podremos hacer cosas como estas:
  - `c = "hola mundo"`
  - `c[0]` # h
  - `c[5:]` # mundo
  - `c[::-3]` # hauo

# Tuplas vs Listas

- La diferencia entre las tuplas y las listas que es **las tuplas NO se pueden modificar**, en cambio las listas si se pueden modificar mediante funciones que se comentarán más adelante.
- Las **listas** pertenecen a los tipos llamados **mutables** y las **tuplas** y las **cadenas** son **inmutables** no se pueden modificar.

# Diccionarios

- Los diccionarios, también llamados **matrices asociativas**, deben su nombre a que son **colecciones** que relacionan una **clave** y un **valor**.
- **Son mutables se pueden modificar.**
- Por ejemplo, un diccionario de películas y directores:  
d = {  
    "Love Actually ": "Richard Curtis",  
    "Kill Bill": "Tarantino",  
    "Amélie": "Jean-Pierre Jeunet"  
}

# Diccionarios

- La diferencia principal entre los diccionarios y las listas o las tuplas es que a los valores almacenados en un diccionario se les accede no por su índice, porque de hecho no tienen orden, sino por su clave, utilizando de nuevo el operador [].

`d["Love Actually "] # devuelve "Richard Curtis"`

- Al igual que en listas y tuplas también se puede utilizar este operador para reasignar valores.

`d["Kill Bill"] = "Quentin Tarantino"`

# Conjuntos

- Son como las listas, pero no se pueden modificar y tampoco admiten repetidos.
- Son secuencias pero no se pueden modificar.
- No soportan indexación [] como las listas, tuplas o diccionarios.
- Se pueden utilizar para quitar repetidos en una lista.

# Constructores

- Estos constructores crearían secuencias vacías:
  - Para las listas: **list()** o podemos utilizar: **[]**
  - Para las tuplas: **tuple()** o valores separados por **comas** (entre paréntesis pero no es obligatorio).
  - Para los diccionarios: **dict()** o **{}**
    - Utilizar la sintaxis adecuada para inicializar.
  - Para los conjuntos: **set()**
    - Si queremos dar valores al set → {1,2,3,4,56,8}

# Clasificación de tipos

- **Simples:**
  - Enteros, reales, complejos, bool, NoneType.
  - No son iterables.
  - Son inmutables.
- **Iterables** (son secuencias):
  - Cadenas, conjuntos, listas, diccionarios, tuplas.
- **Inmutables:**
  - Cadenas, tuplas y conjuntos.
- **Mutables:**
  - Listas y diccionarios (listas admiten repetidos, diccionarios NO)
- **Indexables** (con []):
  - Todas las secuencias menos los conjuntos.
  - Podemos también utilizar **slicing** (menos en los conjuntos).



# Cadenas

- Las cadenas de caracteres pueden ir entre comillas dobles o simples.
- 'Esto es una cadena en python'
- El operador + en las cadenas concatena.  
"hola" + "adios" → "holaadios"
- El operador \* en las cadenas, repite tantas veces:  
"hola" \* 3 → "holaholahola"
- Con el operador in, podemos determinar si una cadena se encuentra contenida en otra.

# Secuencias de escape

<code>\a</code>	Bell
<code>\b</code>	Espacio atrás
<code>\f</code>	Alimentación de formulario.
<code>\r</code>	Retorno de carro.
<code>\t</code>	Tabulador horizontal.
<code>\v</code>	Tabulador vertical.
<code>\\</code>	Escapar la barra.
<code>\'</code>	Escapar la comilla simple.
<code>\"</code>	Escapar las dobles comillas.

# Cadenas

- Las cadenas se pueden indexar con el operador []. Indicando el índice: de 0 a len(cadena)-1
- Recorrido:  
for car in cadena:  
 print(car)
- Operador de substring: [ini:fin]
  - a = 'ejemplo'
  - a[2:5] → emp

# Índices

- Índices en las cadenas.

Permite índices positivos y negativos						
0	1	2	3	4	5	6
E	J	E	M	P	L	O
-7	-6	-5	-4	-3	-2	-1

# Índices

- Para obtener 'emp' de la anterior:  
a[2:5], a[-5,5], a[2:-2], a[-5:-2]
- Corte por:  
a[0:j]      →      a[:j]  
a[j:]      →      a[j:len(a)]  
a[:]      →      a
- También se puede hacer con :: indicando 3 parámetros:  
c[0:len(c):2] sólo los pares.

# Métodos de diccionarios, cadenas, listas, conjuntos

# Diccionarios

- **A partir de un diccionario: D**
- **D[k]**
  - Devuelve el valor de la clave k. Si no existe la clave k, devuelve una excepción.
- **D.get(k[, d])**
  - Busca el valor de la clave k en el diccionario. Es equivalente a utilizar D[k] pero al utilizar este método ***podemos indicar un valor a devolver por defecto si no se encuentra la clave***, mientras que con la sintaxis D[k], de no existir la clave se lanzaría una excepción. En este caso devuelve “d”.
- **D.items()**
  - Devuelve un objeto **dict\_keys** de tuplas con pares clave-valor. **Se puede iterar** por este objeto utilizando un **for**. Si queremos una lista como en py 2.7, utilizar el constructor **list**.

# Diccionarios

- **D.keys()**
  - Devuelve un objeto dict\_keys de las claves del diccionario. Se puede iterar con for y pasar a una lista con list.
- **D.pop(k[, d])**
  - Borra la clave k del diccionario y devuelve su valor. Si no se encuentra dicha clave se devuelve d si se especificó el parámetro o bien se lanza una excepción.
- **D.values()**
  - Devuelve un objeto dict\_values con los valores del diccionario. Se puede iterar con for y pasar a una lista con list



# Diccionarios

- Para localizar una clave dentro de un diccionario no se puede utilizar el método **has\_key(clave)**.
- Utilizar el operador **IN**.
- if clave in diccionario:  
    pass

# Diccionarios

- **dict.fromkeys(secuencia, valor)**
  - Se puede aplicar directamente a dict o a un objeto de tipo dict.
  - Crea un nuevo diccionario utilizando la secuencia para crear tantas claves con el valor indicado en el segundo parámetro.
  - `print(dict.fromkeys("ABC","123"))`
  - `print(dict.fromkeys([1,2,3,4,5],"123"))`
- **dict.clear()**
  - Limpia el diccionario, lo vacía.

# Diccionarios

- **dict.update(otro\_dict)**
  - Actualiza un diccionario a partir de otro.  
dict = {'Name': 'Zara', 'Age': 7}  
dict2 = {'Sex': 'female' }  
dict.update(dict2)  
print ("Value : %s" % dict)
- **dict.copy()**
  - Devuelve una copia del diccionario.
- **dict.popitem()**
  - Devuelve y elimina la última tupla (k, v) del diccionario, excepción si está vacío.
- **del dic[clave]**
  - Borra una clave del diccionario.

-

# Cadenas

- **Cadenas:** Entre comillas simples o dobles.
  - Operadores:
    - + Concatena cadenas.
    - \* cadena y un entero. Repite tantas veces la cadena.  
`'ab' * 2 = 'abab'`

# Métodos Cadenas

- **A partir de una cadena S**
- **S.count(sub[, start[, end]])**
  - Devuelve el número de veces que se encuentra sub en la cadena. *Los parámetros opcionales start y end definen una subcadena en la que buscar.*
- **S.find(sub[, start[, end]])**
  - Devuelve la posición en la que se encontró por primera vez sub en la cadena o -1 si no se encontró. Existe **rfind** para buscar por la derecha.
  - Forma de uso:  
c.find('h')
- **S.index(sub[, start[, end]])**
  - Igual que la anterior pero devuelve una excepción (ValueError) si no la encuentra.
  - Existe una función **rindex**.
- **S.join(sequence)**
  - Devuelve una cadena resultante de concatenar las cadenas de la secuencia sequence separadas por la cadena sobre la que se llama el método.

# Cadenas

- **S.partition(sep)**
  - Busca el separador sep en la cadena y devuelve una tupla con la subcadena hasta dicho separador, el separador en si, y la subcadena del separador hasta el final de la cadena.
  - Si no se encuentra el separador, la tupla contendrá la cadena en si y dos cadenas vacías.
- **S.replace(old, new[, count])**
  - Devuelve una cadena en la que se han reemplazado todas las ocurrencias de la cadena old por la cadena new.
  - Si se especifica el parámetro count, este indica el número máximo de ocurrencias a reemplazar.
- **S.split([sep [,maxsplit]])**
  - Devuelve una lista conteniendo las subcadenas en las que se divide nuestra cadena al dividir las por el delimitador sep.
  - En el caso de que no se especifique sep, se usan espacios. Si se especifica maxsplit, este indica el número máximo de particiones a realizar.

# Cadenas

- **S.lower()**
  - Devuelve S en minúsculas.
- **S.upper()**
  - Devuelve S en mayúsculas.
- **S.capitalize()**
  - Devuelve S en letra capital.

# Cadenas

- **strip, rstrip, lstrip**
  - Quitar espacios de ambos lados:  
`s = " \t a string example\t "`  
`s = s.strip()`
  - Quitar espacios por la derecha:  
`s = s.rstrip()`
  - Quitar espacios por la izquierda:  
`s = s.lstrip()`
  - También puede ser:  
`s = s.strip(' \t\n\r')`



# Listas

- Representan secuencias de datos como ocurre entre las cadenas.
- `[1,2,3]` pueden ser secuencias de números enteros, reales, cadenas.
- Se pueden anidar:  
`[1,2,3,[4,5], 6]`
- O tener operaciones dentro:
  - `[1, 1+1, 6/2]`
  - **La lista vacía se representa por `[]`**

# Listas

- A las listas se puede aplicar la función len.
  - **len([1,2,3])** → 3
  - **len([])** → 0
  - Podemos utilizar los operadores:
    - + Concatena.  $[1,2] + [3,4] \rightarrow [1,2,3,4]$
    - \* Repite.  $[1,2] * 3 \rightarrow [1,2,1,2,1,2]$
- [] para indexar., empiezan en cero.
- También se puede hacer:  $[1,2,3,4][0] \rightarrow 1$
- Se pueden utilizar índices negativos.

# Listas

- Se puede invertir una lista con: `a[::-1]`
- Se pueden recorrer con un bucle for:

```
for i in [1,2,3,4,5]:  
    print(i)
```

- Con un rango podemos generar listas:

```
for i in range(1,4):  
    print (i)
```

# Listas

- Podemos comparar las listas con los operadores relacionales: `==`, `<`, `<=`, `>`, `>=`
- Tenemos el operador **is** para representar la misma zona de memoria.
- `a = [1,2,3]`
- `b = a`
- `a` y `b` son referencias a los mismos datos.
- El operador **is** devolvería **True**.

# Listas

- El operador `[]` también se puede utilizar para **modificar** los elementos de una lista.

`a[0] = 10`

- **Eliminar** elementos de una lista:

`a = [1,2,3]`

`del a[1]`

`a (enter) # en una consola`

`[1,3]`

# Listas

- También se puede utilizar **el operador in** para comprobar si un elemento pertenece a una lista. Al igual que ocurre en las cadenas.

**#operador in en listas:**

```
numero = 4
```

```
lista = [1,2,3,4,5,6,7]
```

```
if numero in lista:
```

```
    print ('Se encuentra dentro de la lista')
```

```
else:
```

```
    print ('el numero no se encuentra en la lista')
```

# Conversión

- Se pueden convertir cadenas en listas y viceversa.
- `'uno dos tres'.split()` # por defecto el separador es el espacio en blanco.  
`['uno', 'dos', 'tres']`
- Se pueden concatenar los elementos:  
`' '.join(['uno', 'dos', 'tres'])`  
`':'.join(['1','2','3']) → '1:2:3'`
- **list:** Devuelve una lista formada por caracteres individuales a la cadena:  
`list('cadena') → ['c','a','d','e','n','a']`

# Resumen de métodos en Listas

- A partir de una lista L
- **L.append(object)**
  - Añade un objeto al final de la lista.
- **L.count(value)**
  - Devuelve el número de veces que se encontró value en la lista.
- **L.extend(iterable)**
  - Añade los elementos del iterable a la lista.
- **L.index(value[, start[, stop]])**
  - Devuelve la posición en la que se encontró la primera ocurrencia de value. Si se especifican, start y stop definen las posiciones de inicio y fin de una sublista en la que buscar.



# Listas

- **L.insert(index, object)**
  - Inserta el objeto object en la posición index.
- **L.pop([index])**
  - Devuelve el valor en la posición index y lo elimina de la lista. Si no se especifica la posición, se utiliza el último elemento de la lista.
- **L.remove(value)**
  - Eliminar la primera ocurrencia de value en la lista.
- **L.reverse()**
  - Invierte la lista. Esta función trabaja sobre la propia lista desde la que se invoca el método, no sobre una copia.

# Listas

- **L.clear()**
  - Borra todos los elemento de la lista.
- **L.copy()**
  - Devuelve una copia (nueva e independiente del original) de todos los elementos de la lista.

# Listas

- **L.sort(key=None, reverse=False) → None**
  - **Ordena la lista.**
    - Se modifica la lista actual (mutable).
    - **SOLO VALE para LISTAS (no ordena diccionarios, etc).**
    - No devuelve nada.
    - El parámetro **reverse** es un booleano que indica si se debe ordenar la lista de forma inversa, lo que sería equivalente a llamar primero a L.sort() y después a L.reverse().
    - Por último, si se especifica, el parámetro **key** debe ser una función que tome un elemento de la lista y devuelva una clave a utilizar a la hora de comparar, en lugar del elemento en si.

# Ejemplo

- El parámetro Key representa una función que podemos utilizar para ordenar, la función se llama en cada comparación.
- Esta función sólo puede recibir un parámetro y en función de este se ordena.
- Ordenar según el tamaño de cada palabra:  

```
s = "This is a test string from Andrew"  
L = s.split(" ")  
L.sort(key=len)  
print(L)
```

# **sorted(iterable, key=None,reverse=False)**

- Es una función que devuelve una nueva lista ordenada.
- Ejemplo:  

```
>>> sorted([5, 2, 3, 1, 4])  
[1, 2, 3, 4, 5]
```
- A diferencia del método `sort()` de las listas, esta función vale también para diccionarios.  

```
>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})  
[1, 2, 3, 4, 5]
```

# sorted(iterable, key=None, reverse=False)

- Con el parámetro reverse podemos indicar si queremos una ordenación ascendente o descendente.
- Por defecto, ordena ascendente.

- Ejemplo:

```
>>> student_tuples = [  
... ('john', 'A', 15),  
... ('jane', 'B', 12),  
... ('dave', 'B', 10),  
... ]
```

```
>>> sorted(student_tuples, key=lambda student: student[2]) # Ordena  
    por el tercer campo de la tupla.
```

```
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

# Ejemplo I

- Se pueden ordenar clases:
- **class Student:**
  - **def \_\_init\_\_(self, name, grade, age):**
    - self.name = name
    - self.grade = grade
    - self.age = age
  - **def \_\_repr\_\_(self):**
    - **return** repr((self.name, self.grade, self.age))

# Ejemplo II

- `student_objects = [`
  - `Student('john', 'A', 15),`
  - `Student('jane', 'B', 12),`
  - `Student('dave', 'B', 10),`
- `]`
- `sorted(student_objects, key=lambda student: student.age)`
- El módulo `operator` nos facilita el acceso a las propiedades o a los elementos de la tupla.



# Ejemplo III

- **from operator import itemgetter, attrgetter**
- `sorted(student_tuples, key=itemgetter(2))`
- `sorted(student_objects, key=attrgetter('age'))`
- Incluso se pueden especificar varios niveles de ordenamiento:
  - `sorted(student_tuples, key=itemgetter(1,2))`
  - `sorted(student_objects, key=attrgetter('grade', 'age'))`

# Conjuntos

- En los conjunto no podemos indexar, ni hacer slicing, pero si podemos utilizar algunos métodos:
  - Operador **in**, para saber si tiene o no un elemento.
  - Podemos añadir con **add**.
  - Eliminar con **remove**.
  - **copy** para copiar conjuntos.
  - **issuperset**: para saber si un conjunto engloba a otro.
  - Se pueden **inicializar** con una **lista** o con una **cadena**.
    - Quitará los repetidos.
    - Si lo hacemos con un diccionario, tomará las claves.
- Operadores:
  - Diferencia, **|** Unión, **&** Intersección, **^** diferencia simétrica.

# Ejemplo

```
lista = ['vino', 'cerveza', 'agua', 'vino'] # define lista
bebidas = set(lista) # define conjunto a partir de una lista
print('vino' in bebidas) # True, 'vino' está en el conjunto
print('anis' in bebidas) # False, 'anis' no está en el conjunto
print(bebidas) # imprime {'agua', 'cerveza', 'vino'}
bebidas2 = bebidas.copy() # crea nuevo conjunto a partir de copia
print(bebidas2) # imprime {'agua', 'cerveza', 'vino'}
bebidas2.add('anis') # añade un nuevo elemento
print(bebidas2.issuperset(bebidas)) # True, bebidas es un subconjunto
bebidas.remove('agua') # borra elemento
print(bebidas & bebidas2) # imprime elementos comunes
```

```
tapas = ['croquetas', 'solomillo', 'croquetas'] # define lista
conjunto = set(tapas) # crea conjunto (sólo una de croquetas)
if 'croquetas' in conjunto: # evalúa si croquetas está en conjunto
    conjunto1 = set('Python') # define conjunto: P, y, t, h, o, n
    conjunto2 = set('Pitonisa') # define conjunto: P, i, t, o, n, s, a
    print(conjunto2 - conjunto1) # aplica diferencia: s, i, a
    print(conjunto1 | conjunto2) # aplica unión: P, y, t, h, o, n, i, s, a
    print(conjunto1 & conjunto2) # aplica intersección: P, t, o, n
    print(conjunto1 ^ conjunto2) # aplica diferencia simétrica: y, h, i, s, a
```