

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

05.X - Python

Tipos estructurados

Indice

- [Introducción](#)
- [Arrays](#)
- [Tuplas](#)
- [Listas](#)
- [Conjuntos \(Sets\)](#)
- [Diccionarios](#)

```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self,addr,port,func):
        try:
            self.sock.connect((addr,port))
            self.handle=self.sock
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
    def listen(self,host,port,func):
        try:
            self.sock.bind((host,port))
            self.sock.listen(2)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error:Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x,ho=self.sock.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
            def send(self, data):
                self.handle.send(data)
            def close(self):
                self.flag=0
            self.sock.close()
```

Introducción (I)

- Hasta el momento, todo el tratamiento computacional se ha realizado sobre datos de tipos primitivos (int, float, boolean) y un tipo particular de **colección**, las cadenas de texto (String)
- De igual modo que las funciones nos proporcionan un mecanismo para agrupar código en una única “entidad” e invocarlo cuando lo necesitemos, los lenguajes de programación suelen disponer de estructuras de datos que nos permiten agrupar **colecciones** de datos (de igual o diferente tipo) y tratarlos de modo conjunto.
- Python proporciona los siguientes tipos estructurados:

	ARRAY	TUPLA	LISTA	SET	DICCIONARIO
ORDENADO (ind)	X	X	X		
NO ORDENADO				X	X
MUTABLE	X		X	X (sin rep.)	X
INMUTABLE		X			

Introducción (II)

- Varios de los tipos estructurados para el manejo de colecciones son de tipo **secuencia** (array, lista, tupla, string). Cada elemento de la colección se encuentra en una posición concreta (índice) dentro de la secuencia.
- Operaciones comunes de secuencias (tipos mutables e inmutables):

Operación	Resultado
<code>x [not] in s</code>	True si <code>x</code> [no] es igual a un <i>item</i> de <code>s</code> . False en otro caso
<code>s + t</code>	Concatenación de las secuencias <code>s</code> y <code>t</code>
<code>s * n</code>	Equivalente a añadir <code>s</code> a sí mismo <code>n</code> veces
<code>s[i]</code>	Item en la posición <code>i</code> de la secuencia, con origen en 0
<code>s[i:j]</code>	Porción de la secuencia desde <code>i</code> hasta <code>(j-1)</code>
<code>s[i:j:k]</code>	Porción de la secuencia desde <code>i</code> hasta <code>(j-1)</code> con salto <code>k</code>
<code>len(s)</code>	Longitud de <code>s</code>
<code>min(s) max(s)</code>	Valores menor y mayor de <code>s</code>
<code>s.index(x[, i[, j]])</code>	Índice de la primera ocurrencia de <code>x</code> en <code>s</code> (desde <code>i</code> hasta <code>j</code>)(<code>i,j</code> no impl. siempre)
<code>s.count(x)</code>	Número total de ocurrencias del valor <code>x</code> en la secuencia <code>s</code>

Introducción (III)

- Operaciones comunes de secuencias de tipos **mutables**:

Operación	Resultado
<code>s[i] = x</code>	El <i>item</i> <code>i</code> es reemplazado por <code>x</code>
<code>s[i:j] = t</code>	La porción de <code>s</code> desde <code>i</code> hasta <code>j</code> es reemplazada por el contenido del iterable <code>t</code>
<code>del s[i]</code>	Elimina el <i>item</i> en la posición <code>i</code>
<code>del s[i:j]</code>	Elimina los <i>items</i> desde <code>i</code> hasta <code>(j-1)</code>
<code>del s[i:j:k]</code>	Elimina los <i>items</i> desde <code>i</code> hasta <code>(j-1)</code> y salto <code>k</code>
<code>s.append(x)</code>	Añade <code>x</code> al final de la secuencia <code>s</code>
<code>s.clear()</code>	Elimina todos los <i>items</i> de <code>s</code> (no disponible en array)
<code>s.copy()</code>	Crea una copia de <code>s</code>
<code>s.insert(i, x)</code>	Inserta <code>x</code> en la secuencia <code>s</code> en la posición de índice <code>i</code>
<code>s.pop()</code>	Elimina de la secuencia el último <i>item</i> . Devuelve el valor eliminado
<code>s.pop(i)</code>	Elimina de la secuencia el <i>item</i> en la posición <code>i</code> . Devuelve el valor eliminado
<code>s.remove(x)</code>	Elimina de la secuencia la primera ocurrencia de <code>x</code>
<code>s.reverse()</code>	Invierte el orden de la secuencia

Introducción (y IV)

- Una característica de estas estructuras de datos de tipo secuencia, al igual que otros objetos de tipo contenedor como los *String*, es que soportan la **iteración** a través de sus elementos.
- Esto nos permiten recorrerlos fácilmente mediante bucles *for*

```
>>> for element in [1, 2, 3]:  
...     print(element)  
>>> for element in (1, 2, 3):  
...     print(element)  
>>> for key in {'one':1, 'two':2}:  
...     print(key)  
>>> for char in {'123'}:  
...     print(char)  
>>> for line in open("myfile.txt"):  
...     print(line, end='')
```

Arrays (I)

- Los **arrays** son el método tradicional de crear **colecciones** de datos primitivos, donde todos los datos de la colección, son del **mismo tipo**. Si bien son populares en lenguajes como C, C++ o Java, no lo son tanto en Python, pues dispone de tipos estructurados más flexibles (listas).
- En general, cuando la gente habla de arrays en Python, suelen referirse a las listas. Sin embargo, son estructuras de datos diferentes. En Python, los arrays pueden ser vistos como una manera **eficiente** de almacenar cierta listas, donde todos los elementos son del mismo tipo.
- En Python, el soporte de arrays lo proporciona el **módulo array** que debe ser **importado** antes de poder inicializarlos y usarlos.
- Durante la **creación** del array, deberemos indicar su **tipo** de datos, **limitando** de esta manera el **rango** y tipo de los valores almacenados
- Documentación oficial: <https://docs.python.org/3/library/array.html>

Arrays (II)

CÓDIGO TIPO	TIPO C EQUIV	TIPO PYTHON	TAMAÑO(Bytes)	RANGO VALORES
'b'	signed char	int	1	$[-2^7, 2^7-1]$ $[-128, 127]$
'B'	unsigned char	int	1	$[0, 2^8-1]$ $[0, 255]$
'u'	wchar_t	Unicode	2	'\u0000' - '\uFFFF'
'h'	signed short	int	2	$[-2^{15}, 2^{15}-1]$ $[-32.768, 32.767]$
'H'	unsigned short	int	2	$[0, 2^{16}-1]$ $[-65.536, 65.535]$
'i'	signed int	int	4(2*)	$[-2^{31}, 2^{31}-1]$ $[-2.147.483.648, 2.147.483.647]$
'I'	unsigned int	int	4(2*)	$[0, 2^{32}-1]$ $[0, 4.294.967.296]$
'l'	signed long	int	8(4*)	$[-2^{63}, 2^{63}-1]$ $[-9.223.372.036.854.775.808,$ $9.223.372.036.854.775.807]$
'L'	unsigned long	int	8(4*)	$[0, 2^{64}-1]$ $[0, 18.446.744.073.709.551.615]$
'f'	float	float	4	$[1.2E-38, 3.4E+38]$
'd'	double	float	8	$[2.3E-308, 1.7E+308]$

Tabla de *códigos de tipo* para la creación de arrays en Python (*dep. arquitectura)

Arrays (y III). Ejemplos

```
>>> import array as arr
>>> udata = arr.array('B', (0, 18, 128))
>>> udata
array('B', [0, 18, 128])
>>> udata.append(255)
>>> udata
array('B', [0, 18, 128, 255])
>>> udata.append(300)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    udata.append(300)
OverflowError: unsigned byte integer is greater than maximum
>>> udata[1]
18
>>> chars = arr.array('u', 'hola')
>>> udata.append('\u0061')
>>> chars
array('u', 'holaa')
>>> chars.count('a')
2
```

creación de un array de bytes

adición de elementos al array

error al tratar de añadir valores fuera del rango permitido

count() devuelve el número de ocurrencias en el array del *item* indicado

Tuplas (I)

- Las **tuplas** son secuencias **inmutables**, generalmente empleadas para el almacenamiento **ordenado** de datos **heterogéneos**.
- Su diferencia fundamental con las **listas** es que, una vez creada, no puede ser modificada, a diferencia de las listas. Suelen emplearse para almacenar secuencias de valores constantes (por ejemplo, las claves de un diccionario) o en retornos de funciones de varios valores.
- Las tuplas pueden ser construidas de diferentes maneras:
 - Usando un par de **paréntesis** para denotar la **tupla vacía**: `()`
 - Usando una **coma final** para una tupla simple: `a`, ó `(a,)`
 - Separando los *items* con **comas**: `a, b, c` ó `(a, b, c)`
 - Usando el constructor **tuple()**: `tuple()` ó `tuple(iterable)`
- En realidad es la coma, no el paréntesis, lo que determina a una tupla. Son opcionales salvo en casos ambiguos (paso parámetros a función,...)

Tuplas (II). Ejemplos

```
>>> t = (2, 'uno', 3)
```

← creación de una **tupla**

```
>>> t
```

```
(2, 'uno', 3)
```

```
>>> t[0]
```

← acceso a los elementos mediante **índice**

```
2
```

```
>>> (2, 'uno', 3) + (5, 6)
```

← creación de una nueva tupla por **concatenación**

```
(2, 'uno', 3, 5, 6)
```

```
>>> t[1:3]
```

← extracción de **segmentos** (tuplas) de la tupla

```
('uno', 3)
```

```
>>> type(t[1:3])
```

```
<class 'tuple'>
```

```
>>> t[2:3]
```

```
(3,)
```

```
>>> type(t[2:3])
```

```
<class 'tuple'>
```

```
>>> t[1] = 4
```

← error al tratar de **modificar** un elemento de la tupla

```
Traceback (most recent call last):
```

```
  File "<pyshell#10>", line 1, in <module>
```

```
    t[1] = 4
```

```
TypeError: 'tuple' object does not support item assignment
```

Tuplas (y III). Ejemplos

- Las tuplas se emplean para **intercambiar** el valor de variables:

```
>>> x = 5
>>> y = 6
>>> x = y
>>> y = x
>>> x
6
>>> y
6
```



```
>>> x = 5
>>> y = 6
>>> temp = y
>>> y = x
>>> x = temp
>>> x
6
>>> y
5
```



```
>>> x = 5
>>> y = 6
>>> x, y = y, x
>>> x
6
>>> y
5
```



- Para **retornar más de un valor** desde una función:

```
def div_entera ( x, y ):
    c == x // y
    r == x % y
    return c, r
```

c,r es una tupla

cociente, resto
es una tupla

```
cociente, resto = div_entera(5, 4)
```

Listas (I)

- Como las tuplas, las **listas** son secuencias ordenadas de datos del mismo o diferente tipo. La principal diferencia es que las listas son **mutables**. Es decir, una vez creadas, pueden ser modificadas.
- Este hecho aporta una gran flexibilidad. Por otro lado, puede dar lugar a situaciones indeseables. Por ejemplo, una lista pasada a una función, podría ver modificado su contenido en el interior de la misma.
- Las listas pueden ser construidas de diferentes maneras:
 - Usando un par de **corchetes** para denotar la **lista vacía**: `[]`
 - Usando **corchetes**, separando los *items* con comas: `[a]`, ó `[a, b, c]`
 - Usando listas **por comprensión**: `[x for x in iterable]`
 - Usando el constructor **list()**: `list()` ó `list(iterable)`
- Además de los métodos y operaciones de las secuencias mutables, las listas disponen también del método **sort()**

Listas (II). Indexado

- Como el resto de secuencias ordenadas, podemos acceder a los elementos de la lista mediante un **índice**:

```
>>> lista_1 = []  
>>> lista_2 = [2, 'a', 4, True]  
>>> lista_2  
[2, 'a', 4, True]  
>>> len(lista_1)  
0  
>>> lista_2[0]  
2  
>>> lista_2[2] + 1  
5  
>>> lista[7]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: list index out of range  
>>> i = 2  
>>> lista_2[i-1]  
'a'
```

← creación de listas

← obtención de la longitud de la lista

← acceso a los elementos mediante índice

← error al tratar de acceder fuera del rango del índice de la lista

Listas (II). Ejemplos

- Disponemos de diferentes mecanismos para **añadir**, **modificar** y **eliminar** elementos de la lista:

```
>>> lista_1 = [1, 2]
>>> lista_2 = lista_1 + [2, 'a', 4]
>>> lista_2
```

creación de una nueva
lista por **concatenación**

```
[1, 2, 2, 'a', 4]
>>> lista_2.append(lista_1)
>>> lista_2
```

adición de elementos a la lista

```
[1, 2, 2, 'a', 4, [1, 2]]
>>> len(lista_2)
6
```

```
>>> lista_2[3] = 9
>>> lista_2[5][1] = 4
>>> lista_2
```

acceso y **modificación** de los elementos
de la lista mediante **índice**

```
[1, 2, 2, 9, 4, [1, 4]]
>>> lista_2.sort()
```

error por tratar de ordenar una lista
de elementos **heterogénea**

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: '<' not supported between instances of 'list' and 'int'
```

Listas (III). Métodos *append()* y *extend()*

- Diferencia entre los métodos **append()** y **extend()**

```
>>> lista = [1, 2]
>>> num = 5
>>> tupla = (3, 4)
>>> cadena = 'hola'
>>> lista.append(num)
>>> lista.append(tupla)
>>> lista.append(cadena)
>>> lista
[1, 2, 5, (3, 4), 'hola']
```

append() añade los nuevos elementos, manteniendo su **tipo**, al final de la lista

```
>>> lista = [1, 2]
>>> lista.extend(tupla)
>>> lista.extend(cadena)
>>> lista
[1, 2, 3, 4, 'h', 'o', 'l', 'a']
```

extend() añade de forma individual cada uno de los elementos del *iterable*

```
>>> lista.extend(num)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

error por tratar de **extender** la lista con un objeto no *iterable*

Listas (IV). Métodos *del()*, *remove()* y *pop()*

- Diferencia entre los métodos *del()*, *remove()* y *pop()*

```
>>> lista = [x for x in range(1, 11)]
```

← creación de una lista por comprensión

```
>>> lista
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> del lista[2]
```

```
>>> lista
```

```
[1, 2, 4, 5, 6, 7, 8, 9, 10]
```

← *del()* permite eliminar *items* concretos o *segmentos* enteros identificados mediante *índices*

```
>>> del lista[6:]
```

```
>>> lista
```

```
[1, 2, 4, 5, 6, 7]
```

```
>>> lista.pop()
```

```
7
```

← *pop()* elimina el *último* *item* de la lista o el correspondiente al *índice* indicado. *Devuelve* el valor eliminado

```
>>> lista.pop(1)
```

```
2
```

```
>>> lista
```

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

```
>>> lista.remove(4)
```

```
>>> lista
```

← *remove()* elimina de la lista la *primera* ocurrencia en la lista del *valor* indicado. Genera un *error* si valor no se encuentra en la lista

```
[1, 5, 6]
```

Listas (V). Conversión Listas-Strings

- Conversiones entre listas y cadenas de caracteres:
 - *list(s)*, crea una lista a partir de cada carácter de la cadena *s*
 - *s.split(sep)*, trocea la cadena *s* en subcadenas utilizando *sep* como separador (espacio si no se indica) y genera una lista
 - *s.join(lista)* genera una cadena de caracteres concatenando los elementos (tipo *str*) de *lista* con la cadena *s*

```
>>> s = "hola qué tal?"
>>> list(s)
['h', 'o', 'l', 'a', ' ', 'q', 'u', 'é', ' ', 't', 'a', 'l', '?']
>>> s.split()
['hola', 'qué', 'tal?']
>>> list(s.split()[0])
>>> s.split('a')
['hol', ' qué t', 'l?']
>>> '/' .join(['15', '02', '2018'])
'15/02/2018'
```

Listas (V). Ordenación

- Ordenación de listas. Los elementos deben ser del **mismo tipo** primitivo para poder ordenarlos con los operadores < >
 - El método **sort()** ordena la lista (la modifica)
 - La función **sorted(lista)**, devuelve una lista ordenada a partir de **lista** (no la modifica)
 - El método **reverse()** invierte el orden de la lista (la modifica)

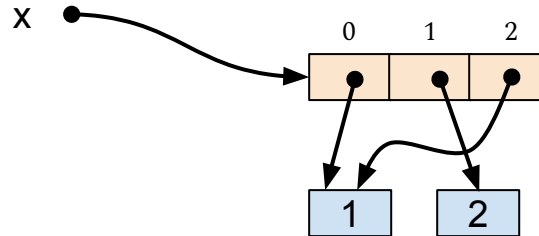
```
>>> lista = [3, 2, 1, 8, 2]
>>> sorted(lista)
[1, 2, 2, 3, 8]
>>> lista
[3, 2, 1, 8, 2]
>>> lista.sort()
>>> lista
[1, 2, 2, 3, 8]
>>> lista.reverse()
>>> lista
[8, 3, 2, 2, 1]
```

Listas (VI). Mutabilidad y Clonado

- Las listas son **objetos mutables** y esto puede tener **efectos colaterales** cuando trabajamos con ellas
- Las listas se implementan como **objetos en memoria** y las variables de tipo lista contienen **referencias** que apuntan a dichos objetos
- Cuando asignamos una variable de tipo lista a otra variable, no se realiza una copia de dicha listas, sino que ambas contendrán la misma referencia al mismo objeto en memoria (son **alias** de la misma lista).
- De lo anterior se desprende que, si **modificamos** una de las listas anteriores, la otra también se verá modificada. En realidad, hay una única lista a la que apuntan ambas variables
- Esto mismo ocurre cuando pasamos una lista como argumento a una función. No se pasa una copia de la lista, sino una referencia a la misma.

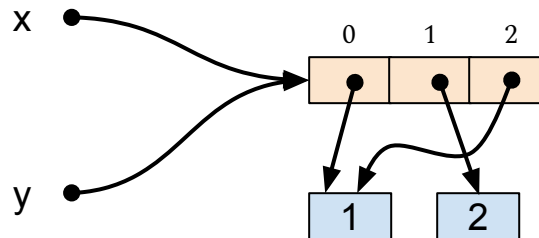
Listas (VII). Mutabilidad y Clonado

`x = [1, 2, 1]`



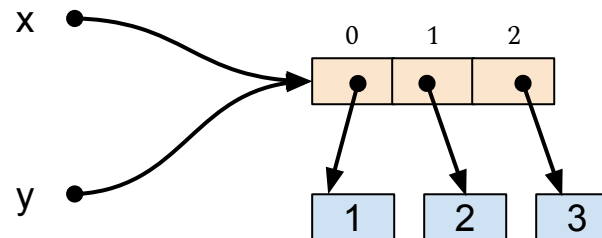
`x[2] → 1`

`y = x`



`x[2] → 1`
`y[2] → 1`

`x[2] = 3`



`x[2] → 3`
`y[2] → 3`

Listas (VIII). Mutabilidad y Clonado

- Cuando lo que queremos es obtener una copia de una lista, podemos:
 - Crear una función que recorra la lista y devuelva una lista con la copia de los valores

```
def copia_lista (lista):  
    copia = []  
    for item in lista: copia += [item]  
    return copia
```

- Emplear el método *copy()* para obtener una nueva lista

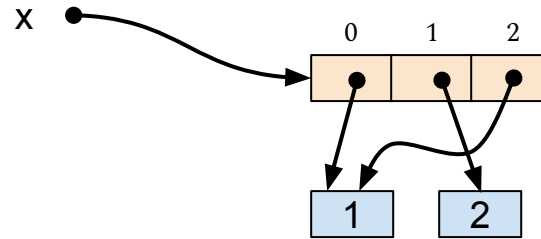
```
>>> lista = [1, 2, 3]  
>>> copia = lista.copy()
```

- Emplear el *operador de indexado* para obtener una nueva lista

```
>>> lista = [1, 2, 3]  
>>> copia = lista[:]
```

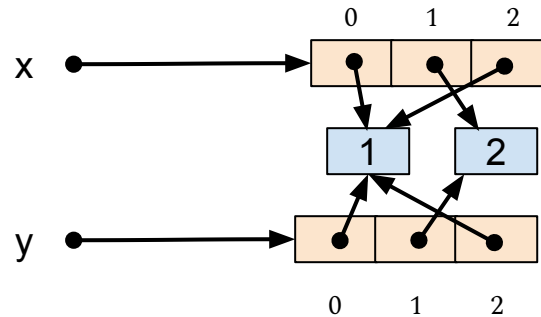
Listas (IX). Mutabilidad y Clonado

`x = [1, 2, 1]`



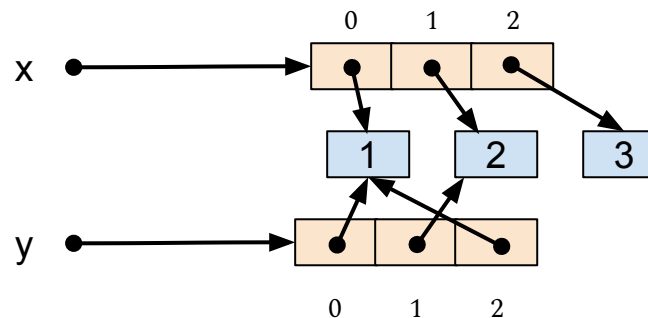
`x[2] → 1`

`y = x[:]`



`x[2] → 1`
`y[2] → 1`

`x[2] = 3`



`x[2] → 3`
`y[2] → 1`

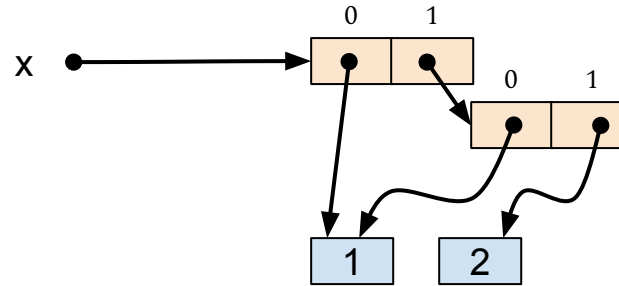
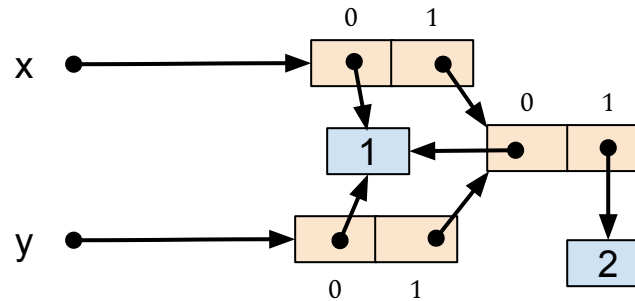
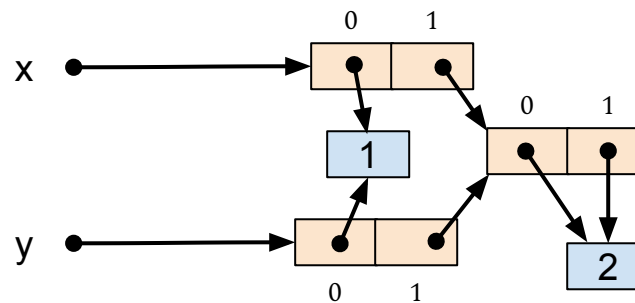
Listas (X). *Swallow vs Deep copy*

- Los ejemplos de copia de listas vistos hasta el momento son ejemplos de lo que se denomina *swallow copy* (copia superficial). Dado que las listas pueden almacenar cualquier tipo de dato, podríamos definir listas contenidas en otras listas. ¿Qué ocurre al modificarlas?

```
>>> colores = [['azul', 'verde'], ['rojo', 'naranja']]
>>> colores_copia = colores[:]
>>> colores_copia
[['azul', 'verde'], ['rojo', 'naranja']]
>>> del colores[0]
>>> colores
[['rojo', 'naranja']]
>>> colores_copia
[['azul', 'verde'], ['rojo', 'naranja']]
>>> del colores_copia[1][0]
>>> colores_copia
[['azul', 'verde'], ['naranja']]
>>> colores
[['naranja']]
```




¡¡ SE MODIFICA
LA LISTA
ORIGINAL !!

Listas (XI). *Swallow vs Deep copy*`x = [1, [1, 2]]``x[1] → [1, 2]``y = x[:]`
`x[1] → [1, 2]`
`y[1] → [1, 2]`
`x[1][0] = 2`
`x[1] → [2, 2]`
`y[1] → [2, 2]`
swallow copy (copia superficial)

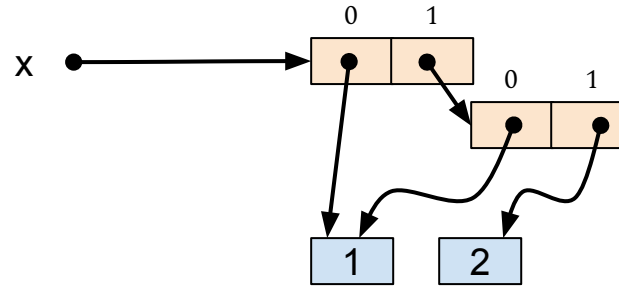
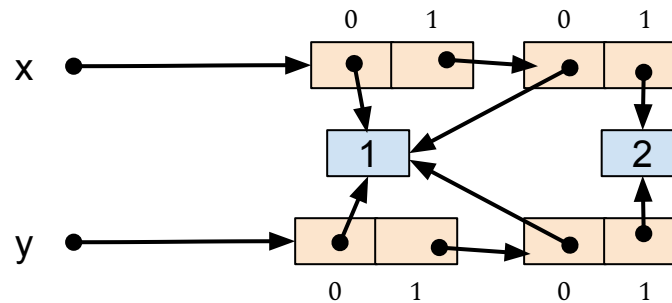
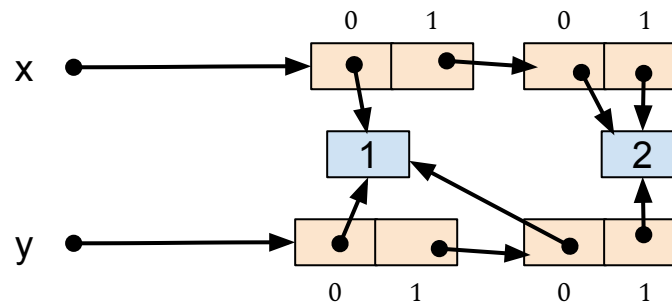
Listas (XI). *Swallow vs Deep copy*

- Cuando tratamos con objetos compuestos, Python nos proporciona funciones para realizar una *deep copy* (copia profunda). Esto es, se construye un nuevo objeto compuesto y, recursivamente, se insertan en él copias de los objetos encontrados en el original

```
>>> import copy
>>> colores = [['azul', 'verde'], ['rojo', 'naranja']]
>>> colores_copia = copy.deepcopy(colores)
>>> colores_copia
[['azul', 'verde'], ['rojo', 'naranja']]
>>> del colores[0]
>>> colores
[['rojo', 'naranja']]
>>> colores_copia
[['azul', 'verde'], ['rojo', 'naranja']]
>>> del colores_copia[1][0]
>>> colores_copia
[['azul', 'verde'], ['naranja']]
>>> colores
[['rojo', 'naranja']]
```



LOS CAMBIOS
DE UNA LISTA
NO AFECTAN A
LA OTRA

Listas (XII). *Swallow vs Deep copy*`x = [1, [1, 2]]``x[1] → [1, 2]``y = copy.deepcopy(x)`
`x[1] → [1, 2]`
`y[1] → [1, 2]`
`x[1][0] = 2`
`x[1] → [2, 2]`
`y[1] → [1, 2]`
deep copy (copia profunda)

Sets (I)

- Los **sets** o **conjuntos**, son secuencias de elementos que cumplen:
 - **No están ordenados**. No soporta indexado.
 - Cada elemento es **único** (no se permiten duplicados)
 - El set es **mutable** pero sus elementos tienen que ser de tipo **inmutable**
- Usos habituales de los sets son su empleo como claves de diccionarios, eliminación de duplicados y el cálculo de operaciones matemáticas del álgebra de conjuntos (unión, intersección, diferencia, complemento,...)
- Al igual que el resto de colecciones, soporta el operador pertenencia (**in**), la función **len()** y su recorrido mediante bucles **for**.
- Los sets pueden ser contruidos de diferentes maneras:
 - Usando **llaves**, separando los *items* con **comas**: **{1}**, ó **{a, b, c}**
 - Usando el constructor **set()**: **set()** ó **set(iterable)**

Sets (II). Ejemplos

```
>>> set_1 = {}  
>>> set_2 = {2, 'a', (1, 4), True}  
>>> set_3 = set('hola')
```

creación de sets

```
>>> set_2  
{2, 'a', (1, 4), True}
```

```
>>> set_3  
{'h', 'o', 'l', 'a'}
```

```
>>> len(set_1)  
4
```

obtención de la longitud del set

```
>>> s1.add(3)  
>>> s1.add('hola')
```

añadir elementos al set mediante add

```
>>> s1.add(3)  
>>> s1
```

el orden puede ser diferente al que fueron añadidos
y no se añaden duplicados de elementos ya existentes

```
{'hola', 3}  
>>> s1.pop()  
'hola'
```

borrado de elementos del set mediante pop, discard y remove

```
>>> s1.discard(3)  
>>> s1.remove(3)
```

Si el elemento a borrar no existe, **remove**
genera una **excepción** de tipo **KeyError**.
discard no genera excepción si no existe

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 3
```

Sets (III)

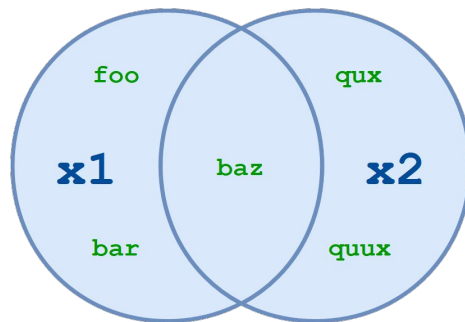
- Operaciones de álgebra de conjuntos:

Operación	Ejemplo
<code>x1.union(x2[, x3 ...])</code> <code>x1 x2 [x3 ...]</code>	<code>>>> {1, 2, 3} {3, 6} {8}</code> <code>{1, 2, 3, 6, 8}</code>
<code>x1.intersection(x2[, x3 ...])</code> <code>x1 & x2 [& x3 ...]</code>	<code>>>> {1, 2, 3} & {3, 6}</code> <code>{3}</code>
<code>x1.difference(x2[, x3 ...])</code> <code>x1 - x2 [- x3 ...]</code>	<code>>>> {1, 2, 3} - {3, 6}</code> <code>{1, 2}</code>
<code>x1.symmetric_difference(x2)</code> <code>x1 ^ x2 [^ x3 ...]</code>	<code>>>> {1, 2, 3} ^ {3, 6}</code> <code>{1, 2, 6}</code>
<code>x1.isdisjoint(x2)</code>	<code>>>> {1, 2, 3}.isdisjoint({3, 6})</code> <code>False</code>
<code>x1.issubset(x2)</code> <code>x1 <= x2</code>	<code>>>> {2, 1} <= {1, 2, 3}</code> <code>True</code>
<code>x1 < x2</code>	<code>>>> {2, 1} < {1, 2, 3}</code> <code>True</code>
<code>x1.issuperset(x2)</code> <code>x1 >= x2</code>	<code>>>> {2, 1, 3} >= {1, 2}</code> <code>True</code>
<code>x1 > x2</code>	<code>>>> {2, 1, 3} > {1, 2}</code> <code>True</code>

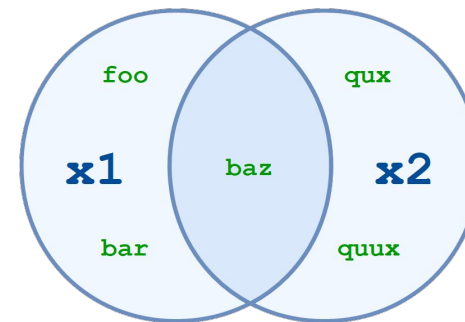
Sets (y IV)

```
x1 = {'foo', 'bar', 'baz'}  
x2 = {'baz', 'qux', 'quux'}
```

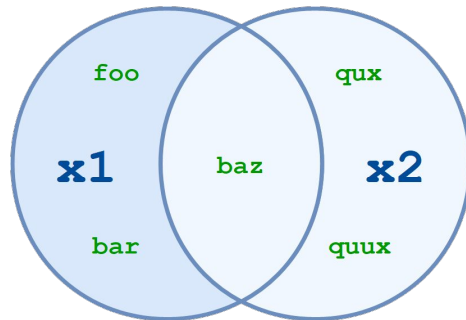
`x1.union(x2)`
`x1 | x2`



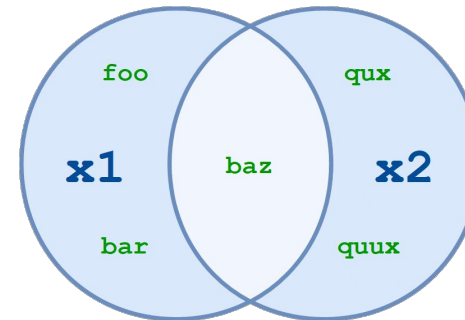
`x1.intersection(x2)`
`x1 & x2`



`x1.difference(x2)`
`x1 - x2`



`x1.symmetric_difference(x2)`
`x1 ^ x2`



Diccionarios (I)

- Los **diccionarios** son estructuras de datos del tipo **arrays asociativos**. Consisten en colecciones **no ordenadas** y **mutables** de **pares *clave:valor***, donde cada clave referencia (*indexa*) un valor concreto. El acceso a los diferentes valores almacenados en el diccionario se realiza a través de la clave correspondiente.
- Las agendas o los diccionarios de las lenguas, son típicos ejemplos de diccionarios. También lo sería cualquier tabla cuyos registros estuvieran identificados por un campo clave, índice o ***primary key***
- Las claves son **únicas** (no puede haber claves duplicadas) y deben ser de un tipo **inmutable** (tipos primitivos y tuplas, si no contienen tipos mutables)
- No hay restricciones en cuanto al tipo de datos de los valores. Además, distintas claves pueden tener valores de diferentes tipos

Diccionarios (II)

- Los diccionarios pueden ser contruidos de diferentes formas:
 - Usando **llaves** para denotar un diccionario vacío: `{ }`
 - Usando llaves, separando las parejas **clave:valor** mediante comas:
`{ 1:'uno', 2:'dos', 'otra_clave':'otro_valor' }`
 - Usando el constructor: **`dict()`** ó **`dict(secuencia de parejas)`**
- Para añadir nuevas parejas **clave:valor** al diccionario, utilizaremos:
`diccionario[nueva_clave] = nuevo_valor`
Si la clave ya existía, se actualizará su valor al nuevo valor proporcionado
- Para eliminar entradas del diccionario, podemos usar la función **`del()`** o el método **`pop()`**, indicando la clave del elemento a eliminar. Ambas generan una excepción **`KeyError`** si la clave no existe.
- El método **`clear()`** elimina todas las entradas de un diccionario

Diccionarios (III). Ejemplos

```
>>> dict_1 = {}  
>>> dict_2 = {1:'uno', 2:'dos', 'a':(1, 2, 3)}  
>>> dict_2  
{1:'uno', 2:'dos', 'a':(1, 2, 3)}  
>>> dict_3 = dict([('G':6.6742e-11), ('pi':3.1416)])  
>>> dict_2  
{1:'uno', 2:'dos', 'a':(1, 2, 3)}  
>>> dict_3['c']=299792458  
>>> dict_3  
{'G':6.6742e-11, 'pi':3.1416, 'c':299792458}  
>>> dict_2['a']  
(1, 2, 3)  
>>> dict_2['a'][2]  
3  
>>> dict_2['a']='letra_a'  
>>> dict_2.get('a')  
'letra_a'  
>>> del dict_2['a']  
>>> dict_2.pop(1)  
>>> dict_2  
{2:'dos'}
```

creación de
diccionarios

adición de elementos al diccionario

acceso y modificación de los elementos

eliminación de elementos

Diccionarios (IV). Unión de diccionarios

- El método `update()` permite fusionar los contenidos de dos diccionarios. Al invocar `d1.update(d2)`, el diccionario `d1` incorporará todas las entradas del diccionario `d2`. Si alguna de las claves de `d2` ya se encuentra en `d1`, ésta se actualizará con el valor de `d2`

```
>>> d1 = {1:'uno', 2:'dos', 3:'tressss'}
>>> d2 = {4:'cuatro', 3:'tres'}
>>> d1.update(d2)
>>> d1
{1: 'uno', 2: 'dos', 3: 'tres', 4: 'cuatro'}
```

- Desde Python 3.5, se dispone de una nueva sintaxis (operador `**`) para generar **nuevos** diccionarios a partir de la **unión** de otros (PEP 448)

```
>>> d1 = {1:'uno', 2:'dos', 3:'tressss'}
>>> d2 = {4:'cuatro', 3:'tres'}
>>> d3 = {**d1, **d2, 5:'cinco', 6:'seis'}
>>> d3
{1: 'uno', 2: 'dos', 3: 'tres', 4: 'cuatro', 5: 'cinco', 6: 'seis'}
```

Diccionarios (V). Vistas

- Los métodos *items()*, *keys()* y *values()* nos proporcionan objetos tipo vista muy útiles a la hora de recorrer y acceder a los valores de los diccionarios
- *items()* devuelve las parejas *clave:valor* contenidas en el diccionario

```
>>> d = {1:'uno', 2:'dos', 3:'tres'}
>>> d.items()
dict_items([(1, 'uno'), (2, 'dos'), (3, 'tres')])
>>> list(d.items())
[(1, 'uno'), (2, 'dos'), (3, 'tres')]
>>> list(d.items())[1][1]
'dos'
>>> for k,v in d.items():
...     print(k,v)
1 uno
2 dos
3 tres
```

Diccionarios (y VI). Vistas

- *keys()* devuelve las **claves** contenidas en el diccionario
- *values()* devuelve una vista de los **valores** en el diccionario

```
>>> d = {1:'uno', 2:'dos', 3:'tres'}
>>> d.keys()
dict_keys([1, 2, 3])
>>> 4 in d.keys()
False
>>> d.values()
dict_values(['uno', 'dos', 'tres'])
>>> list(d.values())
['uno', 'dos', 'tres']
>>> 'uno' in d.values()
True
>>> for k in d.keys():
...     print(k, d[k])
1 uno
2 dos
3 tres
```