TIPOS DE DATOS SECUENCIALES.

Los tipos de datos secuencia me permiten guardar una sucesión de datos de diferentes tipos. Los tipos de datos secuencias en python son:

- Las listas (`list`): Me permiten guardar un conjunto de datos que se pueden repetir y que pueden ser de distintos tipos. Es un tipo mutable.
- Las tuplas (`tuple`): Sirven para lo mismo que las listas, pero en este caso es un tipo inmutable.
- Los rangos (`range`): Es un tipo de secuencias que nos permite crear secuencias de números. Es un tipo inmutable y se suelen utilizar para realizar bucles.
- Las cadenas de caracteres (`str`): Me permiten guardar secuencias de caracteres. Es un tipo inmutable.
- Las secuencias de bytes (`byte`): Me permite guardar valores binarios representados por caracteres ASCII. Es un tipo inmutable.
- Las secuencias de bytes (`bytearray`): En este caso son iguales que las anteriores, pero son de tipo mutables.
- Los conjuntos (`set`): Me permiten guardar conjuntos de datos, en los que no se existen repeticiones. Es un tipo mutable.
- Los conjuntos (`frozenset`): Son iguales que los anteriores, pero son tipos inmutables.

CARACTERÍSTICAS PRINCIPALES DE LAS SECUENCIAS.

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

- Las secuencias se pueden recorrer

```
>>> for num in lista:
...     print(num,end="")
123456
```

- Operadores de pertenencia: Se puede comprobar si un elemento pertenece o no a una secuencia con los operadores `in` y `not in`.

```
>>> 2 in lista
True
>>> 8 not in lista
True
```

- Concatenación: El operador `+` me permite unir datos de tipos secuenciales. No se pueden concatenar secuencias de tipo `range` y de tipo conjunto.

```
>>> lista + [7,8,9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Repetición: El operador `*` me permite repetir un dato de un tipo secuencial. No se pueden repetir secuencias de tipo `range` y de tipo conjunto.

```
>>> lista * 2
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

- Indexación: Puedo obtener el dato de una secuencia indicando la posición en la secuencia. Los conjuntos no tienen implementado esta característica.

```
>>> lista[3]
4
```

- Slice (rebanada): Puedo obtener una subsecuencia de los datos de una secuencia. En los conjuntos no puedo obtener subconjuntos. Esta característica la estudiaremos más detenidamente en la unidad que estudiemos las listas.

```
>>> lista[2:4]
[3, 4]
>>> lista[1:4:2]
[2, 4]
```

- Con la función `len` puedo obtener el tamaño de la secuencia, es decir el número de elementos que contiene.

```
>>> len(lista)
6
```

- Con las funciones `max` y `min` puedo obtener el valor máximo y mínimo de una secuencia.

```
>>> max(lista)
6
>>> min(lista)
1
```

Además en los tipos mutables puedo realizar las siguientes operaciones:

- Puedo modificar un dato de la secuencia indicando su posición.

```
>>> lista[2]=99
>>> lista
[1, 2, 99, 4, 5, 6]
```

- Puedo modificar un subconjunto de elementos de la secuencia haciendo slice.

```
>>> lista[2:4]=[8,9,10]
>>> lista
[1, 2, 8, 9, 10, 5, 6]
```

- Puedo borrar un subconjunto de elementos con la instrucción `del`.

```
>>> del lista[5:]
>>> lista
[1, 2, 8, 9, 10]
```

- Puedo actualizar la secuencia con el operador `*=`

```
>>> lista*=2
>>> lista
[1, 2, 8, 9, 10, 1, 2, 8, 9, 10]
```

LISTAS

Las listas (`list`) me permiten guardar un conjunto de datos que se pueden repetir y que pueden ser de distintos tipos. Es un tipo mutable.

Construcción de una lista

Para crear una lista puedo usar varias formas:

- Con los caracteres `[` y `]`:

```
>>> lista1 = []
>>> lista2 = ["a",1,True]
```

- Utilizando el constructor `list`, que toma como parámetro un dato de algún tipo secuencia.

```
>>> lista3 = list()
>>> lista4 = list("hola")
>>> lista4
['h', 'o', 'l', 'a']
```

Operaciones básicas con listas

Como veíamos en el apartado "Tipo de datos secuencia" podemos realizar las siguientes operaciones:

- Las secuencias se pueden recorrer.
- Operadores de pertenencia: `in` y `not in`.
- Concatenación: `+`
- Repetición: `*`
- Indexación: Cada elemento tiene un índice, empezamos a contar por el elemento en el índice 0. Si intento acceder a un índice que corresponda a un elemento que no existe obtenemos una excepción `IndexError`.

```
>>> lista1[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module
IndexError: list index out of range
```

Se pueden utilizar índices negativos:

```
>>> lista[-1]
6
```

- Slice: Veamos como se puede utilizar
  - `lista[start:end]` # Elementos desde la posición start hasta end-1
  - `lista[start:]` # Elementos desde la posición start hasta el final
  - `lista[:end]` # Elementos desde el principio hata la posición end-1
  - `lista[:]` # Todos Los elementos
  - `lista[start:end:step]` # Igual que el anterior pero dando step saltos.

    Se pueden utilizar también índices negativos, por ejemplo: `lista[::-1]`

Funciones predefinidas que trabajan con listas

```
>>> lista1 = [20,40,10,40,50]
>>> len(lista1)
5
>>> max(lista1)
50
>>> min(lista1)
10
>>> sum(lista1)
150
>>> sorted(lista1)
[10, 20, 30, 40, 50]
>>> sorted(lista1,reverse=True)
[50, 40, 30, 20, 10]
```

Veamos con más detenimiento la función `enumerate`: que recibe una secuencia y devuelve un objeto enumerado como tuplas:

```
>>> seasons = ['Primavera', 'Verano', 'Otoño', 'Invierno']
>>> list(enumerate(seasons))
[(0, 'Primavera'), (1, 'Verano'), (2, 'Otoño'), (3, 'Invierno')]
>>> list(enumerate(seasons, start=1))
[(1, 'Primavera'), (2, 'Verano'), (3, 'Otoño'), (4, 'Invierno')]
```

Las listas son mutables

Como hemos indicado anteriormente las listas es un tipo de datos mutable. Eso tiene para nostros varias consecuencias, por ejemplo podemos obtener resultados como se los que se muestran a continuación:

```
>>> lista1 = [1,2,3]
>>> lista1[2]=4
>>> lista1
[1, 2, 4]
>>> del lista1[2]
>>> lista1
[1, 2]


>>> lista1 = [1,2,3]
>>> lista2 = lista1
>>> lista1[1] = 10
>>> lista2
[1, 10, 3]
```

¿Cómo se copian las listas?

Por lo tanto si queremos copiar una lista en otra podemos hacerlo de varias formas:

```
>>> lista1 = [1,2,3]
>>> lista2=lista1[:]
>>> lista1[1] = 10
>>> lista2
[1, 2, 3]

>>> lista2 = list(lista1)

>>> lista2 = lista1.copy()
```

Listas multidimensionales

A la hora de definir las listas hemos indicado que podemos guardar en ellas datos de cualquier tipo, y evidentemente podemos guardar listas dentro de listas.

```
>>> tabla = [[1,2,3],[4,5,6],[7,8,9]]
>>> tabla[1][1]
5

>>> for fila in tabla:
...   for elem in fila:
...      print(elem,end="")
...   print()

123
456
789
```

MÉTODOS PRINCIPALES DE LISTAS

Cuando creamos una lista estamos creando un objeto de la clase `list`, que tiene definido un conjunto de métodos:

```
lista.append   lista.copy     lista.extend   lista.insert   lista.remove
lista.sort
lista.clear    lista.count    lista.index    lista.pop      lista.reverse
```

Métodos de inserción: append, extend, insert

```
>>> lista = [1,2,3]
>>> lista.append(4)
>>> lista
[1, 2, 3, 4]

>>> lista2 = [5,6]
>>> lista.extend(lista2)
>>> lista
[1, 2, 3, 4, 5, 6]

>>> lista.insert(1,100)
>>> lista
[1, 100, 2, 3, 4, 5, 6]
```

Métodos de eliminación: pop, remove

```
>>> lista.pop()
6
>>> lista
[1, 100, 2, 3, 4, 5]

>>> lista.pop(1)
100
>>> lista
[1, 2, 3, 4, 5]

>>> lista.remove(3)
>>> lista
[1, 2, 4, 5]
```

Métodos de ordenación: reverse, sort,

```
>>> lista.reverse()
>>> lista
[5, 4, 2, 1]

>>> lista.sort()
>>> lista
[1, 2, 4, 5]

>>> lista.sort(reverse=True)
>>> lista
[5, 4, 2, 1]

>>> lista=["hola","que","tal","Hola","Que","Tal"]
>>> lista.sort()
>>> lista
['Hola', 'Que', 'Tal', 'hola', 'que', 'tal']
>>> lista=["hola","que","tal","Hola","Que","Tal"]
>>> lista.sort(key=str.lower)
>>> lista
['hola', 'Hola', 'que', 'Que', 'tal', 'Tal']
```

Métodos de búsqueda: count, index

```
>>> lista.count(5)
1

>>> lista.append(5)
>>> lista
[5, 4, 2, 1, 5]
>>> lista.index(5)
0
>>> lista.index(5,1)
4
>>> lista.index(5,1,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 5 is not in list
```

Método de copia: copy

```
>>> lista2 = lista1.copy()
```

EJERCICIOS PROPUESTOS DE LISTAS.

1. Realiza un programa que, solicite una lista y devuelva una nueva lista invertida.
2. Dada una lista de cadenas, pide una cadena por teclado e indica si está en la lista, indica cuantas veces aparece en la lista, lee otra cadena y sustituye la primera por la segunda en la lista, y por último borra la cadena de la lista.
3. Dada una lista, realiza un programa que indique si está ordenada o no.
4. Escribe un programa que permita crear dos listas de palabras y que, a continuación, escriba las siguientes listas (en las que no debe haber repeticiones):
    a. **Lista de palabras que aparecen en las dos listas.**
    b. **Lista de palabras que aparecen en la primera lista, pero no en la segunda.**
    c. **Lista de palabras que aparecen en la segunda lista, pero no en la primera.**
5. Escribe un programa que permita crear una lista de varias dimensiones. Solicita la dimensión por teclado. Construye la lista y permite:
    a. Mostrar la lista.
    b. Borrar un elemento de la lista: si el elemento está repetido borrará todas las ocurrencias.
    c. Contar todas las ocurrencias de un elemento en la lista.

TUPLAS

Las tuplas (`tuple`): Sirven para los mismo que las listas (me permiten guardar un conjunto de datos que se pueden repetir y que pueden ser de distintos tipos), pero en este caso es un tipo inmutable.

Construcción de una tupla

Para crear una lista puedo usar varias formas:

- Con los caracteres (y):

```
>>> tupla1 = ()
>>> tupla2 = ("a",1,True)
```

- Utilizando el constructor tuple, que toma como parámetro un dato de algún tipo secuencia.

```
>>> tupla3=tuple()
>>> tuple4=tuple([1,2,3])
```

Empaquetado y desempaquetado de tuplas

Si a una variable se le asigna una secuencia de valores separados por comas, el valor de esa variable será la tupla formada por todos los valores asignados.

```
>>> tuple = 1,2,3
>>> tuple
(1, 2, 3)
```

Si se tiene una tupla de longitud k, se puede asignar la tupla a k variables distintas y en cada variable quedará una de las componentes de la tupla.

```
>>> a,b,c=tuple
>>> a
1
```

Operaciones básicas con tuplas

En las tuplas se pueden realizar las siguientes operaciones:

- Las tuplas se pueden recorrer.
- Operadores de pertenencia: `in` y `not in`.
- Concatenación: `+`
- Repetición: `*`
- Indexación
- Slice

Entre las funciones definidas podemos usar: `len`, `max`, `min`, `sum`, `sorted`.

Las tuplas son inmutables

```
>>> tupla = (1,2,3)
>>> tupla[1]=5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Métodos principales

Métodos de búsqueda: `count`, `index`

```
>>> tupla = (1,2,3,4,1,2,3)
>>> tupla.count(1)
2

>>> tupla.index(2)
1
>>> tupla.index(2,2)
5
```

RANGOS

Los rangos (`range`): Es un tipo de secuencias que nos permite crear secuencias de números. Es un tipo inmutable y se suelen utilizar para realizar bucles.

Definición de un rango. Constructor range

Al crear un rango (secuencia de números) obtenemos un objeto que es de la clase `range`:

```
>>> rango = range(0,10,2)
>>> type(rango)
<class 'range'>
```

Veamos algunos ejemplos, convirtirndo el rango en lista para ver la secuencia:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Recorrido de un rango

Los rangos se suelen usar para ser recorrido, cuando tengo que crear un bucle cuyo número de iteraciones lo se de antemanos puedo usar una estructura como esta:

```
>>> for i in range(11):
...     print(i,end=" ")
0 1 2 3 4 5 6 7 8 9 10
```

Operaciones básicas con range

En las tuplas se pueden realizar las siguientes operaciones:

- Los rangos se pueden recorrer.
- Operadores de pertenencia: `in` y `not in`.
- Indexación
- Slice

Entre las funciones definidas podemos usar: `len`, `max`, `min`, `sum`, `sorted`.

Ademas un objeto `range` posee tres atributos que nos almacenan el comienzo, final e intervalo del rango:

```
>>> rango = range(1,11,2)
>>> rango.start
1
>>> rango.stop
11
>>> rango.step
2
```

CADENAS

Las cadenas de caracteres (`str`): Me permiten guardar secuencias de caracteres. Es un tipo inmutable. Como hemos comentado las cadenas de caracteres en python3 están codificada con Unicode.

Definición de cadenas. Constructor str

Podemos definir una cadena de caracteres de distintas formas:

```
>>> cad1 = "Hola"
>>> cad2 = '¿Qué tal?'
>>> cad3 = '''Hola,
    que tal?'''
```

También podemos crear cadenas con el constructor `str` a partir de otros tipos de datos.

```
>>> cad1=str(1)
>>> cad2=str(2.45)
>>> cad3=str([1,2,3])
```

ascii

En los principios de la informática los ordenadores se diseñaron para utilizar sólo caracteres ingleses, por lo tanto se creó una codificación de caracteres, llamada ascii (American Standard Code for Information Interchange) que utiliza 7 bits para codificar los 128 caracteres necesarios en el alfabeto inglés. Posteriormente se extendió esta codificación para incluir caracteres no ingleses. Al utilizar 8 bits se pueden representar 256 caracteres. De esta forma para codificar el alfabeto latino aparece la codificación ISO-8859-1 o Latín 1.

Unicode

La codificación unicode nos permite representar todos los caracteres de todos los alfabetos del mundo, en realidad permite representar más de un millón de caracteres, ya que utiliza 32 bits para su representación, pero en la realidad sólo se definen unos 110.000 caracteres.

UTF-8

UTF-8 es un sistema de codificación de longitud variable para Unicode. Esto significa que los caracteres pueden utilizar diferente número de bytes.

Operaciones básicas con cadenas de caracteres

Como veíamos en el apartado "Tipo de datos secuencia" podemos realizar las siguientes operaciones:

- Las cadenas se pueden recorrer.
- Operadores de pertenencia: `in` y `not in`.
- Concatenación: `+`
- Repetición: `*`
- Indexación
- Slice

Entre las funciones definidas podemos usar: `len`, `max`, `min`, `sorted`.

Las cadenas son inmutables

```
>>> cad = "Hola que tal?"
>>> cad[4]="."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Comparación de cadenas

Las cadenas se comparan carácter a carácter, en el momento en que dos caracteres no son iguales se compara alfabéticamente (es decir, se convierte a código unicode y se comparan).

Ejemplos

```
>>> "a">"A"
True
>>> ord("a")
97
>>> ord("A")
65

>>> "informatica">"informacion"
True

>>> "abcde">"abcdef"
False
```

Funciones repr, ascii, bin

- `repr(objeto)`: Devuelve una cadena de caracteres que representa la información de un objeto.

  ```
  >>> repr(range(10))
  'range(0, 10)'
  >>> repr("piña")
  "'piña'"
  ```

  La cadena devuelta por `repr()` debería ser aquella que, pasada a `eval()`, devuelve el mismo objeto.

  ```
  >>> type(eval(repr(range(10))))
  <class 'range'>
  ```

- `ascii(objeto)`: Devuelve también la representación en cadena de un objeto pero en este caso muestra los caracteres con un código de escape . Por ejemplo en ascii (Latin1) la `á` se presenta con `\xe1`.

  ```
  >>> ascii("á")
  "'\\xe1'"
  >>> ascii("piña")
  "'pi\\xf1a'"
  ```

- `bin(numero)`: Devuelve una cadena de caracteres que corresponde a la representación binaria del número recibido.

  ```
  >>> bin(213)
  '0b11010101'
  ```

Métodos de formato

```
>>> cad = "hola, como estás?"
>>> print(cad.capitalize())
Hola, como estás?

>>> cad = "Hola Mundo"
>>> print(cad.lower())
hola mundo

>>> cad = "hola mundo"
>>> print(cad.upper())
HOLA MUNDO

>>> cad = "Hola Mundo"
>>> print(cad.swapcase())
hOLA mUNDO

>>> cad = "hola mundo"
>>> print(cad.title())
Hola Mundo

>>> print(cad.center(50))
                    hola mundo
>>> print(cad.center(50,"="))
====================hola mundo====================

>>> print(cad.ljust(50,"="))
hola mundo========================================
>>> print(cad.rjust(50,"="))
========================================hola mundo

>>> num = 123
>>> print(str(num).zfill(12))
000000000123
```

Métodos de búsqueda

```
>>> cad = "bienvenido a mi aplicación"
>>> cad.count("a")
3
>>> cad.count("a",16)
2
>>> cad.count("a",10,16)
1

>>> cad.find("mi")
13
>>> cad.find("hola")
-1

>>> cad.rfind("a")
21
```

El método `index()` y `rindex()` son similares a los anteriores pero provocan una excepción `ValueError` cuando no encuentra la subcadena.

Métodos de validación

```
>>> cad.startswith("b")
True
>>> cad.startswith("m")
False
>>> cad.startswith("m",13)
True

>>> cad.endswith("ción")
True
>>> cad.endswith("ción",0,10)
False
>>> cad.endswith("nido",0,10)
True
```

Otras funciones de validación: `isalnum()`, `isalpha()`, `isdigit()`, `islower()`, `isupper()`, `isspace()`, `istitle()`,...

Métodos de sustitución

`format`

En la unidad **"Entrada y salida estándar"** ya estuvimos introduciendo el concepto de formateo de la cadenas. Estuvimos viendo que hay dos métodos y vimos algunos ejemplos del *nuevo estilo* con la función predefinida `format()`.

El uso del estilo nuevo es actualmente el recomendado (puedes obtener más información y ejemplos en algunos de estos enlaces: enlace1 y enlace2) y obtiene toda su potencialidad usando el método `format()` de las cadenas. Veamos algunos ejemplos:

```
>>> '{} {}'.format("a", "b")
'a b'
>>> '{1} {0}'.format("a", "b")
'b a'
>>> 'Coordenadas: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordenadas: 37.24N, -115.81W'
>>> '{0:b} {1:x} {2:.2f}'.format(123, 223,12.2345)
'1111011 df 12.23'
>>> '{:^10}'.format('test')
'   test   '
```

Otros métodos de sustitución

```
>>> buscar = "nombre apellido"
>>> reemplazar_por = "Juan Pérez"
>>> print ("Estimado Sr. nombre apellido:".replace(buscar, reemplazar_por))
Estimado Sr. Juan Pérez:

>>> cadena = "   www.eugeniabahit.com   "
>>> print(cadena.strip())
www.eugeniabahit.com
>>> cadena="00000000123000000000"
>>> print(cadena.strip("0"))
123
```

De forma similar `lstrip(["caracter"])` y `rstrip(["caracter"])`.

Métodos de unión y división

```
>>> formato_numero_factura = ("Nº 0000-0", "-0000 (ID: ", ")"
>>> print("275".join(formato_numero_factura))
Nº 0000-0275-0000 (ID: 275)

>>> hora = "12:23"
>>> print(hora.rpartition(":"))
('12', ':', '23')
>>> print(hora.partition(":"))
('12', ':', '23')
>>> hora = "12:23:12"
>>> print(hora.partition(":"))
('12', ':', '23:12')
>>> print(hora.split(":"))
['12', '23', '12']
>>> print(hora.rpartition(":"))
('12:23', ':', '12')
>>> print(hora.rsplit(":",1))
['12:23', '12']


>>> texto = "Linea 1\nLinea 2\nLinea 3"
>>> print(texto.splitlines())
['Linea 1', 'Linea 2', 'Linea 3']
```

Ejercicios cadenas

1. Crea un programa en python que lea una cadena de caracteres y muestre la siguiente información:
    a. La primera letra de cada palabra. Por ejemplo, si recibe Universal Serial Bus, debe devolver USB.
    b. La última letra de cada palabra. Por ejemplo, si recibe Paramount Comedy deberá devolver ty.
    c. Dicha cadena con la primera letra de cada palabra en mayúsculas. Por ejemplo, si recibe república argentina, debe devolver República Argentina.
    d. Las palabras que comiencen con la letra A. Por ejemplo, si recibe Antes de ayer, debe devolver Antes ayer.
2. Escribe dos funciones que dadas dos cadenas de caracteres:
    a. Indique si la segunda cadena es una subcadena de la primera. Por ejemplo, cadena es una subcadena de subcadena.
    b. Devuelva la que sea anterior en orden alfabético. Por ejemplo, si recibe kde y gnome, debe devolver gnome.
3. Escribir un programa en python, que dada una palabra diga si es un palíndromo. Un palíndromo es una palabra, número o frase que se lee igual hacia adelante que hacia atrás. Ejemplo: reconocer.

CONJUNTOS

set

Los conjuntos (`set`): Me permiten guardar conjuntos (desornedos) de datos (a los que se puede calcular una función hash), en los que no existen repeticiones. Es un tipo de datos mutable.

Normalmente se usan para comprobar si existe un elemento en el conjunto, eliminar duplicados y cálculos matématicos, como la intersección, unión, diferencia,...

Definición de set. Constructor set

Podemos definir un tipo `set` de distintas formas:

```
>>> set1 = set()
>>> set1
set()
>>> set2=set([1,1,2,2,3,3])
>>> set2
{1, 2, 3}
>>> set3={1,2,3}
>>> set3
{1, 2, 3}
```

Frozenset

El tipo `frozenset` es un tipo inmutable de conjuntos.

Definición de frozenset. Constructor frozenset

```
>>> fs1=frozenset()
>>> fs1
frozenset()
>>> fs2=frozenset([1,1,2,2,3,3])
>>> fs2
frozenset({1, 2, 3})
```

Operaciones básicas con set y frozenset

De las operaciones que estudiamos en el apartado "Tipo de datos secuencia" los conjuntos sólo aceptan las siguientes:

- Recorrido
- Operadores de pertenencia: `in` y `not in`.

Entre las funciones definidas podemos usar: `len`, `max`, `min`, `sorted`.

Los set son mutables, los frozenset son inmutables

```
>>> set1={1,2,3}
>>> set1.add(4)
>>> set1
{1, 2, 3, 4}
>>> set1.remove(2)
>>> set1
{1, 3, 4}
```

El tipo `frozenset` es inmutable por lo tanto no posee los métodos `add` y `remove`.

Veamos algunos métodos, partiendo siempre de estos dos conjuntos:

```
>>> set1={1,2,3}
>>> set2={2,3,4}

>>> set1.difference(set2)
{1}
>>> set1.difference_update(set2)
>>> set1
{1}

>>> set1.symmetric_difference(set2)
{1, 4}
>>> set1.symmetric_difference_update(set2)
>>> set1
{1, 4}

>>> set1.intersection(set2)
{2, 3}
>>> set1.intersection_update(set2)
>>> set1
{2, 3}

>>> set1.union(set2)
{1, 2, 3, 4}
>>> set1.update(set2)
>>> set1
{1, 2, 3, 4}
```

Veamos los métodos de añadir y eliminar elementos:

```
>>> set1 = set()
>>> set1.add(1)
>>> set1.add(2)
>>> set1
{1, 2}
>>> set1.discard(3)
>>> set1.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
>>> set1.pop()
1
>>> set1
{2}
```

Y los métodos de comprobación:

```
>>> set1 = {1,2,3}
>>> set2 = {1,2,3,4}
>>> set1.isdisjoint(set2)
False
>>> set1.issubset(set2)
True
>>> set1.issuperset(set2)
False
>>> set2.issuperset(set1)
True
```

ITERADORES

Iterador

Un objeto iterable es aquel que puede devolver un iterador. Normalmente las colecciones que hemos estudiados son iterables. Un iterador me permite recorrer los elementos del objeto iterable.

Definición de iterador. Constructor iter

```
>>> iter1 = iter([1,2,3])
>>> type(iter1)
<class 'list_iterator'>
>>> iter2 = iter("hola")
>>> type(iter2)
<class 'str_iterator'>
```

Función next(), reversed()

Para recorrer el iterador, utilizamos la función `next()`:

```
>>> next(iter1)
1
>>> next(iter1)
2
>>> next(iter1)
3
>>> next(iter1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

La función `reversed()` devuelve un iterador con los elementos invertidos, desde el último al primero.

```
>>> iter2 = reversed([1,2,3])
>>> next(iter2)
3
>>> next(iter2)
2
>>> next(iter2)
1
>>> next(iter2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

El módulo itertools

El módulo itertools contiene distintas funciones que nos devuelven iteradores.

Veamos algunos ejemplos:

`count()`: Devuelve un iterador infinito.

```
>>> from itertools import count
>>> counter = count(start=13)
>>> next(counter)
13
>>> next(counter)
14
```

`cycle()`: devuelve una secuencia infinita.

```
>>> from itertools import cycle
>>> colors = cycle(['red', 'white', 'blue'])
>>> next(colors)
'red'
>>> next(colors)
'white'
>>> next(colors)
'blue'
>>> next(colors)
'red'
```

`islice()`: Retorna un iterador finito.

```
>>> from itertools import islice
>>> limited = islice(colors, 0, 4)
>>> for x in limited:
...   print(x)
white
blue
red
white
```

Generadores

Un generador es un tipo concreto de iterador. Es una función que permite obtener sus resultados paso a paso. Por ejemplo, hacer una función que cada vez que la llamemos nos de el próximo número par. Tenemos dos maneras de crear generadores:

1. Realizar una función que devuelva los valores con la palabra reservada `yield`. Lo veremos con profundidad cuando estudiemos las funciones.

2. Utilizando la sintaxis de las "list comprehension". Por ejemplo:

```
>>> iter1 = (x for x in range(10) if x % 2==0)
>>> next(iter1)
0
>>> next(iter1)
2
>>> next(iter1)
4
```

Nota: yield es parecido a return pero pausa el estado de tu función hasta que decidas usarla de nuevo guardando su estado.

Ejercicios de iteradores y generadores

1. Implementa un generador `fibonacci` que produce los diferentes de la secuencia de Fibonacci, que tiene la forma:

   `0, 1, 1, 2, 3, 5, 8, 13, ...`

   Cuyos dos primeros valores son 0 y 1 por definición y el resto se calculan sumando los dos últimos valores de la sucesión.

2. Implementa un generador `primos` que genere todos los números primos. Por eficiencia, el generador debe ir almacenando los primos encontrados hasta el momento en una lista.
3. Implementa un generador `sucesos (probabilidad)` que produce una secuencia infinita de valores booleanos pseudoaletoreos con `probabilidad` de que sean `True`.

```
import random

def sucesos (prob = .5):
    while True:
        yield random.random () < prob

if __name__ == '__main__':
    from itertools import islice
    print list (islice (sucesos (.7), 10))
```