Tipo de datos secuencia

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 los 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 suele 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

Vamos a ver distintos ejemplos partiendo de una lista, que es una secuencia mutable.

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 implementada 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]
```

Tipo de datos secuencia: 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 intentamos 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 hasta 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 son un tipo de datos mutable. Eso tiene para nosotros 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 de listas

1. Lee por teclado números y guárdalo en una lista, el proceso finaliza cuando metamos un número negativo. Muestra el máximo de los números guardados en la lista, muestra los números pares
2. Realizar un programa que, dada una lista, devuelve una nueva lista cuyo contenido sea igual a la original pero invertida. Así, dada la lista [‘Di’, ‘buen’, ‘día’, ‘a’, ‘papa’], deberá devolver [‘papa’, ‘a’, ‘día’, ‘buen’, ‘Di’].

3. Dada una lista de cadenas, pide una cadena por teclado e indica si está en la lista, indica cuántas 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
4. Dado una lista, hacer un programa que indique si está ordenada o no.

Operaciones avanzadas con secuencias

Las funciones que vamos a estudiar en esta unidad nos acercan al paradigma de la programación funcional que también nos ofrece python. La programación funcional es un paradigma de programación declarativa basado en el uso de funciones matemáticas, en contraste con la programación imperativa, que enfatiza los cambios de estado mediante la mutación de variables.

Función map

map(funcion,secuencia): Ejecuta la función enviada por parámetro sobre cada uno de los elementos de la secuencia.

Ejemplo

```
>>> items = [1, 2, 3, 4, 5]
>>> def sqr(x): return x ** 2
>>> list(map(sqr, items))
[1, 4, 9, 16, 25]
```

Función filter

filter(funcion,secuencia): Devuelve una secuencia con los elementos de la secuencia envíada por parámetro que devuelvan True al aplicarle la función envíada también como parámetro.

Ejemplo

```
>>> lista = [1,2,3,4,5]
>>> def par(x): return x % 2==0
>>> list(filter(par,lista))
```

Función reduce

reduce(funcion,secuencia): Devuelve un único valor que es el resultado de aplicar la función á los elementos de la secuencia.

Ejemplo

```
>>> from functools import reduce
>>> lista = [1,2,3,4,5]
>>> def add(x,y): return x + y
>>> reduce(add,lista)
15
```

list comprehension

list comprehension nos proporciona una alternativa para la creación de listas. Es parecida a la función map, pero mientras map ejecuta una función por cada elemento de la secuencia, con esta técnica se aplica una expresión.

Ejemplo

```
>>> [x ** 3 for x in [1,2,3,4,5]]
[1, 8, 27, 64, 125]

>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]

>>> [x + y for x in [1,2,3] for y in [4,5,6]]
[5, 6, 7, 6, 7, 8, 7, 8, 9]
```

Tipo de datos secuencia: 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
```

Tipo de datos secuencia: Rangos

Los rangos (range): Es un tipo de secuencias que nos permite crear secuencias de números. Es un tipo inmutable y se suele 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, convirtiendo 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.

Además 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
```

Codificación de caracteres

Introducción a la codificación de caracteres

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.

La codificación de caracteres en python3

En Python 3.x las cadenas de caracteres pueden ser de tres tipos: Unicode, Byte y Bytearray.

- El tipo unicode permite caracteres de múltiples lenguajes y cada carácter en una cadena tendrá un valor inmutable.
- El tipo byte sólo permitirá caracteres ASCII y los caracteres son también inmutables.
- El tipo bytearray es como el tipo byte pero, en este caso, los caracteres de una cadena si son mutables.

Algo que debe entenderse (e insiste Mark Pilgrim en su libro *Dive into Python*) es que "los bytes no son caracteres, los bytes son bytes; un carácter es en realidad una abstracción; y una cadena de caracteres es una sucesión de abstracciones".

Funciones chr() y ord()

chr(i): Nos devuelve el carácter Unicode que representa el código i.

```
>>> chr(97)
'a'
>>> chr(1004)
'Ϭ'
```

-

ord(c): recibe un carácter c y devuelve el código unicode correspondiente.

```
>>> ord("a")
97
>>> ord("Ϭ")
1004
```

Tipo de datos cadenas de caracteres

- 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 codificadas 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])
```

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 principales de cadenas

Cuando creamos una cadena de caracteres estamos creando un objeto de la clase str, que tiene definido un conjunto de métodos:

cadena.capitalize cadena.isalnum cadena.join cadena.rsplit
cadena.casefold cadena.isalpha cadena.ljust cadena.rstrip
cadena.center cadena.isdecimal cadena.lower cadena.split
cadena.count cadena.isdigit cadena.lstrip cadena.splitlines
cadena.encode cadena.isidentifier cadena.maketrans cadena.startswith
cadena.endswith cadena.islower cadena.partition cadena.strip
cadena.expandtabs cadena.isnumeric cadena.replace cadena.swapcase
cadena.find cadena.isprintable cadena.rfind cadena.title
cadena.format cadena.isspace cadena.rindex cadena.translate
cadena.format_map cadena.istitle cadena.rjust cadena.upper
cadena.index cadena.isupper cadena.rpartition cadena.zfill

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 las 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 de cadenas

1. Crear un programa que lea por teclado una cadena y un carácter, e inserte el carácter entre cada letra de la cadena. Ej: separar y , debería devolver s,e,p,a,r,a,r
2. Crear un programa que lea por teclado una cadena y un carácter, y reemplace todos los dígitos en la cadena por el carácter. Ej: su clave es: 1540 y X debería devolver su clave es: XXXX
3. Crea un programa python que lea una cadena de caracteres y muestre la siguiente información:
    - La primera letra de cada palabra. Por ejemplo, si recibe Universal Serial Bus debe devolver USB.
    - Dicha cadena con la primera letra de cada palabra en mayúsculas. Por ejemplo, si recibe república argentina debe devolver República Argentina.
    - Las palabras que comiencen con la letra A. Por ejemplo, si recibe Antes de ayer debe devolver Antes ayer.

4. Escribir funciones que dadas dos cadenas de caracteres:
    - Indique si la segunda cadena es una subcadena de la primera. Por ejemplo, cadena es una subcadena de subcadena.
    - Devuelva la que sea anterior en orden alfabético. Por ejemplo, si recibe kde y gnome debe devolver gnome.
5. Escribir un programa python que dado 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

Tipo de datos binarios: bytes, bytearray

Bytes

El tipo bytes es una secuencia inmutable de bytes. Sólo admiten caracteres ASCII. También se pueden representar los bytes mediante números enteros cuyo valores deben cumplir 0 <= x < 256.

Definición de bytes. Constructor bytes

Podemos definir un tipo bytes de distintas formas:

```
>>> byte1 = b"Hola"
>>> byte2 = b'¿Qué tal?'
>>> byte3 = b'''Hola,
        que tal?'''
```

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

```
>>> byte1=bytes(10)
>>> byte1
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> byte2=bytes(range(10))
>>> byte2
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t'
>>> byte3=bytes.fromhex('2Ef0 F1f2')
>>> byte3
b'.\xf0\xf1\xf2'
```

Bytearray

El tipo bytearray es un tipo mutable de bytes.

Definición de bytearray. Constructor bytearray

```
>>> ba1=bytearray()
>>> ba1
bytearray(b'')
>>> ba2=bytearray(10)
>>> ba2
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> ba3=bytearray(range(10))
>>> ba3
bytearray(b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t')
>>> ba4=bytearray(b"hola")
>>> ba4
bytearray(b'hola')
>>> ba5=bytearray.fromhex('2Ef0 F1f2')
>>> ba5
bytearray(b'.\xf0\xf1\xf2')
```

Operaciones básicas con bytes y bytearray

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

- Recorrido
- Operadores de pertenencia: in y not in.
- Concatenación: +
- Repetición: *
- Indexación
- Slice

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

Los bytes son inmutables, los bytearray son mutables

```
>>> byte=b"hola"
>>> byte[2]=b'g'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment

>>> ba1=bytearray(b'hola')
>>> ba1[2]=123
>>> ba1
bytearray(b'ho{a')
>>> del ba1[3]
>>> ba1
bytearray(b'ho{')
```

Métodos de bytes y bytearray

byte1.capitalize byte1.index byte1.join byte1.rindex byte1.strip
byte1.center byte1.isalnum byte1.ljust byte1.rjust byte1.swapcase
byte1.count byte1.isalpha byte1.lower byte1.rpartition byte1.title
byte1.decode byte1.isdigit byte1.lstrip byte1.rsplit byte1.translate
byte1.endswith byte1.islower byte1.maketrans byte1.rstrip byte1.upper
byte1.expandtabs byte1.isspace byte1.partition byte1.split byte1.zfill
byte1.find byte1.istitle byte1.replace byte1.splitlines
byte1.fromhex byte1.isupper byte1.rfind byte1.startswith

bytearray1.append bytearray1.index bytearray1.lstrip bytearray1.rstrip
bytearray1.capitalize bytearray1.insert bytearray1.maketrans bytearray1.split
bytearray1.center bytearray1.isalnum bytearray1.partition bytearray1.splitlines
bytearray1.clear bytearray1.isalpha bytearray1.pop bytearray1.startswith
bytearray1.copy bytearray1.isdigit bytearray1.remove bytearray1.strip
bytearray1.count bytearray1.islower bytearray1.replace bytearray1.swapcase
bytearray1.decode bytearray1.isspace bytearray1.reverse bytearray1.title
bytearray1.endswith bytearray1.istitle bytearray1.rfind bytearray1.translate
bytearray1.expandtabs bytearray1.isupper bytearray1.rindex bytearray1.upper
bytearray1.extend bytearray1.join bytearray1.rjust bytearray1.zfill
bytearray1.find bytearray1.ljust bytearray1.rpartition
bytearray1.fromhex bytearray1.lower bytearray1.rsplit

Si nos fijamos la mayoría de los métodos en el caso de los bytes son los de las cadenas de caracteres, y en los bytearray encontramos también métodos propios de las listas.

Métodos encode y decode

Los caracteres cuyo código es mayor que 256 no se pueden usar para representar los bytes, sin embargo sí podemos indicar una codificación de caracteres determinada para que ese carácter se convierta en un conjunto de bytes.

```
>>> byte1=b'piña'
  File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
>>> byte1=bytes('piña',"utf-8")
>>> byte1
b'pi\xc3\xb1a'
>>> len(byte1)
5
>>> byte1=bytes('piña',"latin1")
>>> byte1
b'pi\xf1a'
```

Podemos también convertir una cadena unicode a bytes utilizando el método encode:

```
>>> cad="piña"
>>> byte1=cad.encode("utf-8")
>>> byte1
b'pi\xc3\xb1a'
```

Para hacer la función inversa, convertir de bytes a unicode utilizamos el método decode:

```
>>> byte1.decode("utf-8")
'piña'
```

El problema lo tenemos si hemos codificado utilizando un código e intentamos decodificar usando otro.

```
>>> byte1=bytes('piña',"latin1")
>>> byte1.decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in position 2: invalid continuation byte
>>> byte1.decode("utf-8","ignore")
'pia'
>>> byte1.decode("utf-8","replace")
'pi�a'
```

Tipo de datos conjuntos: set, frozenset

set

Los conjuntos (set): Me permiten guardar conjuntos (desordenados) 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 matemáticos, 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.

Métodos de set y frozenset

set1.add set1.issubset

```
set1.clear                    set1.issuperset
set1.copy                     set1.pop
set1.difference               set1.remove
set1.difference_update        set1.symmetric_difference
set1.discard                  set1.symmetric_difference_update
set1.intersection             set1.union
set1.intersection_update      set1.update
set1.isdisjoint
```

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
```

Por último los métodos de frozenset:

```
fset1.copy              fset1.isdisjoint        fset1.symmetric_difference
fset1.difference        fset1.issubset          fset1.union
fset1.intersection      fset1.issuperset
```

Tipo de datos: iterador y generador

Iterador

Un objeto iterable es aquel que puede devolver un iterador. Normalmente las colecciones que hemos estudiado 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.

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
```

Tipo de datos mapa: diccionario

Los diccionarios son tipos de datos que nos permiten guardar valores, a los que se puede acceder por medio de una clave. Son tipos de datos mutables y los campos no tienen asignado orden.

Definición de diccionarios. Constructor dict

```
>>> a = dict(one=1, two=2, three=3)

>>> b = {'one': 1, 'two': 2, 'three': 3}

>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))

>>> d = dict([('two', 2), ('one', 1), ('three', 3)])

>>> e = dict({'three': 3, 'one': 1, 'two': 2})

>>> a == b == c == d == e

True
```

Si tenemos un diccionario vacío, al ser un objeto mutable, también podemos construir el diccionario de la siguiente manera.

```
>>> dict1 = {}

>>> dict1["one"]=1

>>> dict1["two"]=2

>>> dict1["three"]=3
```

Operaciones básicas con diccionarios

```
>>> a = dict(one=1, two=2, three=3)
```

len(): Devuelve número de elementos del diccionario.

```
>>> len(a)

3
```

Indexación: Podemos obtener el valor de un campo o cambiarlo (si no existe el campo nos da una excepción KeyError):

```
>>> a["one"]
1
>>> a["one"]+=1
>>> a
{'three': 3, 'one': 2, 'two': 2}
```

del():Podemos eliminar un elemento, si no existe el campo nos da una excepción KeyError:

```
>>> del(a["one"])
>>> a
{'three': 3, 'two': 2}
```

Operadores de pertenencia: key in d y key not in d.

```
>>> "two" in a
True
```

iter(): Nos devuelve un iterador de las claves.

```
>>> next(iter(a))
'three'
```

Los diccionarios son tipos mutables

Los diccionarios, al igual que las listas, son tipos de datos mutable. Por lo tanto podemos encontrar situaciones similares a las que explicamos en su momentos con las listas.

```
>>> a = dict(one=1, two=2, three=3)
```

```
>>> a["one"]=2
>>> del(a["three"])
>>> a
{'one': 2, 'two': 2}

>>> a = dict(one=1, two=2, three=3)
>>> b = a
>>> del(a["one"])
>>> b
{'three': 3, 'two': 2}
```

En este caso para copiar diccionarios vamos a usar el método copy():

```
>>> a = dict(one=1, two=2, three=3)
>>> b = a.copy()
>>> a["one"]=1000
>>> b
{'three': 3, 'one': 1, 'two': 2}
```

Métodos principales de diccionarios

dict1.clear dict1.get dict1.pop dict1.update

dict1.copy dict1.items dict1.popitem dict1.values

dict1.fromkeys dict1.keys dict1.setdefault

Métodos de eliminación: clear

```
>>> dict1 = dict(one=1, two=2, three=3)
```

```
>>> dict1.clear()
>>> dict1
{}
```

Métodos de agregado y creación: copy, dict.fromkeys, update, setdefault

```
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict2 = dict1.copy()

>>> dict.fromkeys(["one","two","three"])
{'one': None, 'two': None, 'three': None}
>>> dict.fromkeys(["one","two","three"],100)
{'one': 100, 'two': 100, 'three': 100}

>>> dict1 = dict(one=1, two=2, three=3)
>>> dict2 = {'four':4,'five':5}
>>> dict1.update(dict2)
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}

>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.setdefault("four",4)
4
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> dict1.setdefault("one",-1)
```

```
1
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Métodos de retorno: get, pop, popitem, items, keys, values

```
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.get("one")
1
>>> dict1.get("four")
>>> dict1.get("four","no existe")
'no existe'

>>> dict1.pop("one")
1
>>> dict1
{'two': 2, 'three': 3}
>>> dict1.pop("four")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'four'
>>> dict1.pop("four","no existe")
'no existe'

>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.popitem()
```

('one', 1)

>>> dict1

{'two': 2, 'three': 3}

>>> dict1 = dict(one=1, two=2, three=3)

>>> dict1.items()

dict_items([('one', 1), ('two', 2), ('three', 3)])

>>> dict1.keys()

dict_keys(['one', 'two', 'three'])

El tipo de datos dictviews

Los tres últimos métodos devuelven un objeto de tipo dictviews.

Esto devuelve una vista dinámica del ciccionario, por ejemplo:

>>> dict1 = dict(one=1, two=2, three=3)

>>> i = dict1.items()

>>> i

dict_items([('one', 1), ('two', 2), ('three', 3)])

>>> dict1["four"]=4

>>> i

dict_items([('one', 1), ('two', 2), ('three', 3), ('four', 4)])

Es este tipo de datos podemos usar las siguientes funciones:

- len(): Devuelve número de elementos de la vista.
- iter(): Nos devuelve un iterador de las claves, valores o ambas.
- x in dictview: Devuelve True si x está en las claves o valores.

Recorrido de diccionarios

Podemos recorrer las claves:

```
>>> for clave in dict1.keys():
...     print(clave)
one
two
three
```

Podemos recorrer los valores:

```
>>> for valor in dict1.values():
...     print(valor)
1
2
3
```

O podemos recorrer ambos:

```
>>> for clave,valor in dict1.items():
...     print(clave,"->",valor)
one -> 1
two -> 2
three -> 3
```

Ejercicios de diccionarios

1. Escribe un programa que lea una cadena y devuelva un diccionario con la cantidad de apariciones de cada palabra en la cadena. Por ejemplo, si recibe "Qué lindo día que hace hoy" debe devolver: 'que': 2, 'lindo': 1, 'día': 1, 'hace': 1, 'hoy': 1

2. Tenemos guardado en un diccionario los códigos morse correspondientes a cada carácter. Escribir un programa que lea una palabra y la muestra usando el código morse.
3. Continuar el programa: ahora nos pide un código morse donde cada letra está separada por espacios y nos da la cadena correspondiente.
4. Suponga un diccionario que contiene como clave el nombre de una persona y como valor una lista con todas sus "gustos". Desarrolle un programa que agregue "gustos" a la persona:

- Si la persona no existe la agregue al diccionario con una lista que contiene un solo elemento.
- Si la persona existe y el gusto existe en su lista, no tiene ningún efecto.
- Si la persona existe y el gusto no existe en su lista, agrega el gusto a la lista.

Se deja de pedir personas cuando introducimos el carácter "*".