# Programación en Python
# TEMA 7
## Listas

Universidad Politécnica de Valencia

2021-2022

## Índice

# 1. Una lista es una secuencia

Al igual que una cadena, una **lista** es una secuencia de valores. En una cadena, los valores son caracteres; en una lista, pueden ser de cualquier tipo. Los valores en una lista se llaman **elementos** o a veces **ítems**.

Hay varias maneras de crear una lista nueva; la más simple es encerrar los elementos en corchetes (`[` y `]`):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

El primer ejemplo es una lista de cuatro enteros. El segundo es una lista de tres cadenas. Los elementos de una lista no tienen que ser del mismo tipo. La siguiente lista contiene una cadena, un número de coma flotante, un entero y (¡atención!) otra lista:

```
['spam', 2.0, 5, [10, 20]]
```

Una lista dentro de otra lista está **anidada**.

Una lista que no contiene elementos se llama lista vacía; puedes crear una con corchetes vacíos, `[]`.

Como podrías esperar, puedes asignar valores de lista a variables:

```
>>> quesos = ['Cheddar', 'Edam', 'Gouda']
>>> numeros = [42, 123]
>>> vacio = []
>>> print(quesos, numeros, vacio)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

# 2. Las listas son mutables

La sintaxis para acceder a los elementos de una lista es la misma que para acceder a los caracteres de una cadena: el operador de corchetes. La expresión dentro de los corchetes especifica el índice. Recuerda que los índices comienzan en 0:

```
>>> quesos[0]
'Cheddar'
```

A diferencia de las cadenas, las listas son mutables. Cuando el operador de corchetes aparece en el lado izquierdo de una asignación, este identifica el elemento de la lista que será asignado.

```
>>> numeros = [42, 123]
>>> numeros[1] = 5
>>> numeros
[42, 5]
```

El uno-ésimo elemento de `numeros`, que solía ser 123, ahora es 5.

La Figura 1 muestra el diagrama de estado para `quesos`, `numeros` y `vacio`.


Figura 1: Diagrama de estado.

Las listas se representan por cajas con la palabra "list" por fuera y los elementos de la lista por dentro. `quesos` se refiere a una lista con tres elementos con índices 0, 1 y 2. `numeros` contiene dos elementos; el diagrama muestra que el valor del segundo elemento ha sido reasignado de 123 a 5. `vacio` se refiere a una lista sin elementos.

Los índices de las listas funcionan de la misma manera que los índices de las cadenas:

- Cualquier expresión entera se puede utilizar como índice.
- Si intentas leer o escribir un elemento que no existe, obtienes un `IndexError`.
- Si un índice tiene un valor negativo, se cuenta hacia atrás desde el final de la lista.

El operador `in` también funciona en las listas.

```
>>> quesos = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in quesos
True
>>> 'Brie' in quesos
False
```

## 3. Recorrer una lista

La manera más común de recorrer los elementos de una lista es con un ciclo `for`. La sintaxis es la misma que para las cadenas:

```
for queso in quesos:
    print(queso)
```

Esto funciona bien si solo necesitas leer los elementos de la lista. Pero si quieres escribir o actualizar los elementos, necesitas los índices. Una manera común de hacer eso es combinar las funciones incorporadas `range` y `len`:

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

Este bucle recorre la lista y actualiza cada elemento. `len` devuelve el número de elementos en la lista. `range` devuelve una lista de índices de 0 a $n-1$, donde $n$ es el largo de la lista. En cada paso por el bucle, `i` obtiene el índice del siguiente elemento. La sentencia de asignación en el cuerpo usa `i` para leer el valor antiguo del elemento y asignar el nuevo valor.

Un bucle `for` a través de una lista vacía nunca ejecuta el cuerpo:

```
for x in []:
    print('Esto nunca ocurre.')
```

A pesar de que una lista puede contener otra lista, la lista anidada aún cuenta como un solo elemento. La longitud de esta lista es cuatro:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

## 4. Operaciones de lista

El operador + concatena listas:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

El operador * repite una lista un número dado de veces:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

El primer ejemplo repite `[0]` cuatro veces. El segundo ejemplo repite la lista `[1, 2, 3]` tres veces.

## 5. Trozos de lista

El operador de corte o de trozo que hemos visto para strings, también funciona en las listas:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Si omites el primer índice, el trozo comienza al principio. Si omites el segundo, el trozo llega al final. Entonces, si omites ambos, el trozo es una copia de la lista completa.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Dado que las listas son mutables, a menudo es útil crear una copia antes de realizar operaciones que modifiquen listas.

Un operador de trozo en el lado izquierdo de una asignación puede actualizar múltiples elementos:

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

## 6. Métodos de lista

Python proporciona métodos que operan en listas. Métodos se llaman con la notación del punto, como veremos en el ejemplo abajo `t.append('d')`. Por ejemplo, `append` agrega un nuevo elemento al final de la lista:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

`extend` toma otra lista como argumento y anexa todos los elementos:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

Este ejemplo deja a `t2` sin modificar.

`sort` ordena los elementos de la lista de menor a mayor:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

La mayoría de los métodos de lista son nulos: modifican la lista y devuelven `None`. Si por casualidad escribes `t = t.sort()`, te decepcionará el resultado.

## 7. Recorrer las listas

Para sumar todos los números de una lista, puedes utilizar un bucle como este:

```
def sumar_todos(t):
    total = 0
    for x in t:
        total += x
    return total
```

`total` se inicializa en 0. En cada paso por el bucle, `x` obtiene un elemento de la lista. Recuerda que el operador += proporciona una manera corta de actualizar una variable. Esta **sentencia de asignación aumentada**:

```
total += x
```

es equivalente a

```
total = total + x
```

A medida que el bucle se ejecuta, `total` acumula la suma de los elementos; una variable que se utiliza de esta manera a veces se llama **acumulador**.

Sumar los elementos de una lista es una operación tan común que Python la facilita como función incorporada, **sum**:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

Una operación como esta que combina una secuencia de elementos en un solo valor a veces se llama **reducción**.

A veces quieres recorrer una lista mientras construyes otra. Por ejemplo, la siguiente función toma una lista de cadenas y devuelve una nueva lista que contiene cadenas que comienzan con mayúscula:

```
def todas_con_mayuscula(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

la variable `res` se inicializa con una lista vacía; en cada paso por el bucle, anexamos el elemento siguiente. Entonces `res` es otro tipo de acumulador.

Una recorrido como la en `todas_con_mayuscula` a veces es llamada **mapa** porque "mapea" una función (en este caso el método `capitalize`) sobre cada uno de los elementos en una secuencia.

Otra operación común es seleccionar algunos de los elementos de una lista y devolver una sublista. Por ejemplo, la siguiente función toma una lista de cadenas y devuelve una lista que contiene solo las cadenas escritas con mayúsculas:

```
def solo_mayusculas(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` es un método de cadena que devuelve `True` si la cadena solo contiene letras mayúsculas.

Una recorrido como la en `solo_mayusculas` se llama **filtro** porque selecciona algunos de los elementos y filtra los otros.

La mayoría de las operaciones de lista se pueden expresar como una combinación de mapa, filtro y reducción.

## 8. Eliminar elementos

Hay varias maneras de eliminar elementos de una lista. Si conoces el índice del elemento que quieres, puedes utilizar el método `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

`pop` modifica la lista y devuelve el elemento que se eliminó. Si no entregas un índice, elimina y devuelve el último elemento.

Si no necesitas el valor eliminado, puedes utilizar el operador **`del`**:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

Si conoces el elemento que quieres eliminar (pero no el índice), puedes utilizar el método `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

El valor de retorno de `remove` es `None`.

Para eliminar más de un elemento, puedes utilizar **`del`** con índices de trozo o corte:

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

Como siempre, el trozo selecciona todos los elementos hasta el segundo índice pero sin incluirlo.

## 9. Listas y cadenas

Una cadena es una secuencia de caracteres y una lista es una secuencia de valores, pero una lista de caracteres no es lo mismo que una cadena. Para convertir una cadena en una lista de caracteres, puedes utilizar la función **`list`**:

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

Dado que **`list`** es el nombre de una función predefinida, deberías evitar utilizarlo como nombre de variable. Además es buena practica evitar la letra `l` porque se parece mucho al numero `1`. Entonces por eso usamos aqui `t`.

La función **`list`** separa la cadena en letras individuales. Si quieres separar una cadena en palabras, puedes utilizar el método `split`:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'fjords']
```

Un argumento opcional llamado **delimitador** especifica qué caracteres usar como separador de palabras. El siguiente ejemplo usa un guión como delimitador:

```
>>> s = 'spam-spam-spam'
>>> delimitador = '-'
>>> t = s.split(delimitador)
>>> t
['spam', 'spam', 'spam']
```

`join` es el inverso de `split`. Toma una lista de cadenas y concatena los elementos. `join` es un método de cadena, por lo que tienes que invocarlo en el delimitador y pasarle la lista como parámetro:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimitador = ' '
>>> s = delimitador.join(t)
>>> s
'pining for the fjords'
```

En este caso el delimitador es un carácter de espacio, por lo que `join` pone un espacio entre las palabras. Para concatenar cadenas sin espacios, puedes usar la cadena vacía, `''`, como delimitador.

## 10. Objetos, valores e igualdad

Si ejecutamos estas sentencias de asignación:

```
a = 'banana'
b = 'banana'
```

sabemos que las variables `a` y `b` se refieren a una cadena, pero no sabemos si se refieren a la *misma* cadena. Hay dos estados posibles, mostrados en la Figura 2.


Figura 2: Variables y sus referencias a valores

En el primer caso, `a` y `b` se refieren a dos objetos diferentes que tienen el mismo valor. En el segundo caso, se refieren al mismo objeto.

Para verificar si dos variables se refieren al mismo objeto, puedes usar el operador **`is`**.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
>>> a == b
True
```

En este ejemplo, Python solo crea un objeto de cadena y tanto `a` como `b` se refieren a este. Pero cuando creas dos listas, Python lo hace diferente y nos da dos objetos:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
>>> a == b
True
```

En este caso diríamos que las dos listas son **equivalentes**, porque tienen los mismos elementos, pero no **idénticos**, porque no son el mismo objeto. Si dos objetos son idénticos, son también equivalentes, pero si son equivalentes, no necesariamente son idénticos.

Hasta ahora, hemos estado utilizando "objeto" y "valor" indistintamente, pero es más preciso decir que un objeto tiene un valor. Si evalúas `[1, 2, 3]`, obtienes un objeto de lista cuyo valor es una secuencia de enteros. Si otra lista tiene los mismos elementos, decimos que tiene el mismo valor, pero no es el mismo objeto.

## 11. Alias

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

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

La asociación de una variable con un objeto se llama **referencia**. En este ejemplo, hay dos referencias al mismo objeto.

Un objeto con más de una referencia tiene más de un nombre, por lo que decimos que el objeto tiene un **alias**.

Si el objeto con alias es mutable, los cambios realizados con un alias afectan al otro:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b[0] = 42
>>> a
[42, 2, 3]
```

Aunque este comportamiento puede ser útil, es propenso a errores. En general, es más seguro evitar los alias cuando trabajes con objetos mutables.

Para los objetos inmutables como las cadenas, los alias no son tan problemáticos. En este ejemplo:

```
a = 'banana'
b = 'banana'
```

casi nunca hace una diferencia si `a` y `b` se refieren a la misma cadena o no.

## 12. Argumentos de lista

Cuando pasas una lista como argumento a una función, la función obtiene una referencia a la lista. Si la función modifica la lista, la sentencia llamadora ve el cambio. Por ejemplo, `sin_cabeza` quita el primer elemento de una lista:

```
def sin_cabeza(t):
    del t[0]
```

Se utiliza de la siguiente manera:

```
>>> letras = ['a', 'b', 'c']
>>> sin_cabeza(letras)
>>> letras
['b', 'c']
```

El parámetro `t` y la variable `letras` son alias para el mismo objeto.

Es importante distinguir entre operaciones que modifican listas y operaciones que crean nuevas listas. Por ejemplo, el método `append` modifica una lista, pero el operador `+` crea una nueva lista.

Aquí hay un ejemplo que utiliza `append`:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

El valor de retorno de `append` es `None`.

Aquí hay un ejemplo que utiliza el operador `+`:

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
```

El resultado del operador es una lista nueva y la lista original no ha cambiado.

Esta diferencia es importante cuando escribes funciones que se supone que modifican listas. Por ejemplo, esta función *NO* elimina la cabeza de una lista:

```
def sin_cabeza_mal(t):
    t = t[1:]
```

El operador de trozo crea una nueva lista y la asignación hace que `t` se refiera a esta, pero eso no afecta a la llamadora.

```
>>> t4 = [1, 2, 3]
>>> sin_cabeza_mal(t4)
>>> t4
[1, 2, 3]
```

Al principio de `sin_cabeza_mal`, `t` y `t4` se refieren a la misma lista. Al final, `t` se refiere a una nueva lista, pero `t4` aún se refiere a la original, la lista sin modificar.

Una alternativa es escribir una función que cree y devuelva una nueva lista. Por ejemplo, `cola` devuelve todos los elementos de una lista excepto el primero:

```
def cola(t):
    return t[1:]
```

Esta función deja a la lista original sin modificar. Se utiliza de la siguiente manera:

```
>>> letra = ['a', 'b', 'c']
>>> resto = cola(letras)
>>> resto
['b', 'c']
```

## 13. Depuración

El uso descuidado de las listas (y otros objetos mutables) puede llevar a largas horas de depuración. Aquí hay algunas trampas comunes y maneras de evitarlas:

1. La mayoría de los métodos de lista modifican el argumento y devuelven `None`. Esto es lo opuesto a los métodos de cadena, que devuelven una nueva cadena y dejan sola a la original.

   Si te acostumbraste a escribir código de cadena como este:

   ```
   palabra = palabra.strip()
   ```

   Es tentador escribir código de lista como este:

   ```
   t = t.sort()
   ```

   Pero, dado que `sort` devuelve `None`, es probable que la siguiente operación que realices con `t` falle.

   Antes de utilizar métodos y operadores de lista, deberías leer la documentación cuidadosamente y luego probarlos en modo interactivo.

2. Escoge una forma y quédate con esa.

   Parte del problema con las listas es que hay muchas maneras de hacer las cosas. Por ejemplo, para eliminar un elemento de una lista, puedes utilizar `pop`, `remove`, `del`, o incluso una asignación de trozo.

   Para agregar un elemento, puedes utilizar el método `append` o el pythoninline `+`. Suponiendo que `t` es una lista y `x` es un elemento de lista, estas líneas son correctas:

   ```
   t.append(x)
   t = t + [x]
   t += [x]
   ```

   Y estas son incorrectas:

   ```
   t.append([x])          # ¡INCORRECTO!
   t = t.append(x)        # ¡INCORRECTO!
   t + [x]                # ¡INCORRECTO!
   t = t + x              # ¡INCORRECTO!
   ```

   Prueba cada uno de estos ejemplos en modo interactivo para asegurarte de que entiendes lo que haces. Nota que solo el último provoca un error de tiempo de ejecución; los otros tres son legales, pero hacen lo incorrecto.

3. Crea copias para evitar los alias.

   Si quieres utilizar un método como `sort` que modifique el argumento, pero necesitas mantener la lista original también, puedes crear una copia.

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

En este ejemplo podrías utilizar también la función incorporada **sorted**, que devuelve una nueva lista ordenada y deja sola a la original.

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

## 14. Glosario

**lista:** Una secuencia de valores.

**elemento:** Uno de los valores en una lista (u otra secuencia), también llamados ítems.

**lista anidada:** Una lista que es un elemento de otra lista.

**acumulador:** Una variable utilizada en un bucle para sumar o acumular un resultado.

**asignación aumentada:** Una sentencia que actualiza el valor de una variable utilizando un operador como +=.

**reducción:** Un patrón de procesamiento que recorre una secuencia y acumula los elementos en un solo resultado.

**mapa:** Un patrón de procesamiento que recorre una secuencia y realiza una operación en cada elemento.

**filtro:** Un patrón de procesamiento que recorre una lista y selecciona los elementos que satisfacen algún criterio.

**objeto:** Algo a lo cual una variable puede referirse. Un objeto tiene un tipo y un valor.

**equivalente:** Que tiene el mismo valor.

**idéntico:** Que es el mismo objeto (lo cual implica equivalencia).

**referencia:** La asociación entre una variable y su valor.

**alias:** Una circunstancia donde dos o más variables se refieren al mismo objeto.

**delimitador:** Un carácter o cadena utilizado para indicar dónde debería separarse una cadena.

## Ejercicios

**Exercise 1:**

Escribe una función llamada `nested_sum` que tome una lista de listas de enteros y sume los elementos de todas las listas anidadas. Por ejemplo:

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

Tienes que testear tu programa con pytest. Puedes usar por ejemplo los siguientes casos de test:

```
import pytest
@pytest.mark.parametrize("testcase, entrada, salida_esperada",[
(1, [[2,3,4], [0,0], [-4], [-5, 8]], 8),
(2, [], 0),
(3, [[],[],[]], 0),
(4, [[5]], 5),
(5, [[], [-8]], -8)]
)

def test_nested_sum(testcase, entrada, salida_esperada):
    assert nested_sum(entrada) == salida_esperada, "caso {0}".format(testcase)
```

**Exercise 2:**

Escribe una función llamada `acumular_sum` que tome una lista de números y devuelva la suma acumulativa, es decir, una lista nueva donde el $i$-ésimo elemento es la suma de los primeros $i + 1$ elementos de la lista original. Por ejemplo:

```
>>> t = [1, 2, 3]
>>> acumular_sum(t)
[1, 3, 6]
```

Tienes que testear tu programa con pytest. Puedes usar por ejemplo los siguientes casos de test:

```
@pytest.mark.parametrize("testcase, entrada, salida_esperada",[
(1, [2,2,3,3,4,4,5,5], [2, 4, 7, 10, 14, 18, 23, 28]),
(2, [], []),
(3, [12,2,0,3,0,4,-6,5,15], [12, 14, 14, 17, 17, 21, 15, 20, 35]),
(4, [5], [5]),
(5, [-8,8], [-8,0])]
)

def test_acumular_sum(testcase, entrada, salida_esperada):
    assert acumular_sum(entrada) == salida_esperada, "caso {0}".format(testcase)
```

**Exercise 3:**

Escribe una función llamada `medio` que tome una lista y devuelva una nueva lista que contenga todos los elementos excepto el primero y el último. Por ejemplo:

```
>>> t = [1, 2, 3, 4]
>>> medio(t)
[2, 3]
```

Tienes que testear tu programa con pytest. Puedes usar por ejemplo los siguientes casos de test:

```
@pytest.mark.parametrize("testcase, entrada, salida_esperada",[
(1, [2,2,3,3,4,4,5,5], [2,3,3,4,4,5]),
(2, [], []),
(3, [12], []),
(4, [5,6], []),
(5, [8,8,8], [8])]
)

def test_medio(testcase, entrada, salida_esperada):
    assert medio(entrada) == salida_esperada, "caso {0}".format(testcase)
```

**Exercise 4:**

Escribe una función llamada `acortar` que tome una lista, la modifique eliminando el primer y último elemento, y devuelva `None`. Por ejemplo:

```
>>> t = [1, 2, 3, 4]
>>> acortar(t)
>>> t
[2, 3]
```

Nota que aunque parece mucho a la función `medio` del ejercicio anterior, hay una diferencias a la hora de escribir los pytests. La función no devuelve ningún resultado, pero modifica directamente el argumento. Entonces primero tenemos que llamar la función y después tenemos que chequear que la lista se ha cambiado como esperamos.

```
@pytest.mark.parametrize("testcase, entrada, salida_esperada",[
(1, [2,2,3,3,4,4,5,5], [2,3,3,4,4,5]),
(2, [], []),
(3, [12], []),
(4, [5,6], []),
(5, [8,8,8], [8])]
)

def test_medio(testcase, entrada, salida_esperada):
    t = entrada
    acortar(t)
    assert t == salida_esperada, "caso {0}".format(testcase)
```

**Exercise 5:**

Dos palabras son anagramas si puedes reordenar las letras de una para escribir la otra. Escribe una función llamada `es_anagrama` que tome dos cadenas y devuelva `True` si son anagramas.

En la pagina https://www.ejemplos.co/50-ejemplos-de-anagramas/ puedes encontrar ejemplos de anagramas para escribir tus pytests.

**Exercise 6:**

Escribe una función llamada `tiene_duplicados` que tome una lista y devuelva `True` si hay algún elemento que aparece más de una vez. No debería modificar la lista original. No olvides tus pytests.

**Exercise 7:**

Escribe un modulo `cosas_con_lists` que tiene las siguientes funciones con sus respectivos pytests.

- Una función (`sum_list`) para sumar todos los elementos en la lista.
- Una función (`mult_list`) para multiplicar todos los elementos en la lista.
- Una función (`max_list`) para obtener el numero máximo de la lista
- Una función (`min_list`) para obtener el numero mínimo de la lista

Tu función tiene que pasar los siguientes tests:

```
@pytest.mark.parametrize("testcase, entrada, salida_esperada",[
(1, [0,2,3,0,6,8], 19),   #suma con 0
(2, [0], 0),              #solo 1 elemento
(3, [], 0)])              #lista vacia

def test_sum_list(testcase, entrada, salida_esperada):
    assert sum_list(entrada) == salida_esperada,\
           "caso {0}".format(testcase)

@pytest.mark.parametrize("testcase, entrada, salida_esperada",[
(1, [0,2,3,0,6,8], 0),  #lista con numero 0
(2, [1,2,3,6], 36),     #lista sin numero 0
(3, [1], 1),            #solo 1 elemento
(4, [], "undefined")])           #lista vacia

def test_mult_list(testcase, entrada, salida_esperada):
    assert mult_list(entrada) == salida_esperada,\
           "caso {0}".format(testcase)

@pytest.mark.parametrize("testcase, input, output",[
(1, [0,2,3,0,6,8], 8), #max esta al final
(2, [8,2,3,0], 8),     #max esta al principio
(3, [2,8,3,0], 8),     #max no al principio ni al fin
(4, [8,2,8,3,0], 8),   #max esta repetido
(5, [1,2,3,-6], 3),    #numeros negativos
(6, [1], 1),           #solo 1 elemento
(7, [], "undefined")]) #lista vacia

def test_max_list(testcase, entrada, salida_esperada):
    assert max_num_in_list(entrada) == salida_esperada,\
           "caso {0}".format(testcase)

@pytest.mark.parametrize("testcase, entrada, salida_esperada",[
(1, [1,2,3,6,8,0], 0),   #min esta al final
(2, [0,8,2,3], 0),       #min esta al principio
(3, [2,8,1,5,3], 1),     #min no al principio ni al fin
(4, [8,0,2,3,0], 0),     #min esta repetido
(5, [1,2,3,-6,0,1], -6), #numeros negativos
(6, [1], 1),             #solo 1 elemento
(7, [], "undefined")])   #lista vacia

def test_min_list(testcase, entrada, salida_esperada):
    assert min_num_in_list(entrada) == salida_esperada,\
           "caso {0}".format(testcase)
```

**Exercise 8:**

Escriba una función (`match_words`)) en python que, dado una lista de strings, cuenta el número de cadenas que tiene las siguientes características:

- la longitud de la cadena es mas de 2

- el primer y el último carácter son los mismos

Tu función tiene que pasar los siguientes tests:

```
@pytest.mark.parametrize("testcase, input, output",[
(1, ["aba", "bb", "abc", ""], 1),
(2, ["   ", " a "], 2),
(3, [], 0)])

def test_match_words(testcase, input, output):
    assert match_words(input) == output,\
           "caso {0}".format(testcase)
```

Escribir una función (`mayor_a_media_list`) en Python que dado una lista devuelve una lista en que todos los elementos que son mayor a la media de todos los elementos se han eliminado. Tu función tiene que pasar los siguientes tests:

```
@pytest.mark.parametrize("testcase, input, output",[
(1, [2, 3, 4, 5], [2,3]),
(2, [1,1,1,1],[1,1,1,1] ),
(3, [], []),
(4, [0,0], [0,0]),
(5, [-1,-2,-3], [-2,-3])])

def test_mayor_a_media_list(testcase, input, output):
    assert mayor_a_media_list(input) == output,\
           "caso {0}".format(testcase)
```

**Exercise 9:**

Escribir una función (`sum_of_diagonal`) en python que dado una matriz $m$ de integers calcula la suma de los integers que estan en el diagonal. Por ejemplo:

$$\texttt{sum\_of\_diagonal}(\begin{bmatrix}1 & 2 & 3 & 4\\ 2 & 4 & 6 & 1\\ 0 & 5 & 8 & 2\\ 2 & 9 & 6 & 3\end{bmatrix}) = 16, \quad \texttt{sum\_of\_diagonal}(\begin{bmatrix}1 & 5\\ 3 & 4\end{bmatrix}) = 5$$

Tu función tiene que pasar los siguientes tests:

```
@pytest.mark.parametrize("testcase, input, output",[
(1, [[1,2,3],[4,5,6],[7,8,9]], 15),
(2, [[1,0,1],[1,1,0],[1,1,1]], 3),
(3, [[2,0],[0,2]], 4)])

def test_sum_of_diagonal(testcase, input, output):
    assert sum_of_diagonal(input) == output,\
           "caso {0}".format(testcase)
```