Iteración sobre secuencias mediante `for`

Aviso a navegantes

En esta sección vas a crear muchas funciones que reciben una lista como parámetro.

Es posible que te sientas tentado a llamar su parámetro `list` :

```
def foo(list):
    # cuerpo de la función
    pass
```

¡¡No lo hagas!!

`list` es el tipo de todas las listas y se puede usar para crear otras listas. Si usas ese nombre como tu parámetro, le harás sombra al `list` de Python y te meterás en un lío tarde o temprano.

Algoritmo Universal para Diseñar Funciones 2.0

1. Di en voz alta o escribe lo siguiente:
   A. ¿Qué devuelve esta función? *¿Un número, una cadena, una lista, un booleano?*
   B. ¿Qué recibe esta función? *¿Uno o más parámetros? ¿De qué tipo?*
   C. ¿Hay casos especiales? ¿Qué se hace en ese caso?
2. Escribe la *firma de la función* (su primera linea, del def a los dos puntos) y su documentación.
3. En el cuerpo de la función, pon en comentarios, todos los pasos que crees que harán falta.
4. Cuando estés seguro de entender lo que vas a hacer, empieza a sustituir (a añadir) a cada comentario por la linea de código Python correspondiente.

(1)

Escribe la función `sum_all` que recibe una lista de números y devuelve la suma total.

En este ejercicio, veremos el proceso general que hay que seguir y los demás los harás tú sólo.

Vamos a aplicar el *Algoritmo Universal*:

Entrada y Salida

- `sum_all` devolverá siempre un número
- `sum_all` recibirá siempre una *lista de números*.

Casos especiales

Cuando trabajamos con listas siempre hay un caso especial que hay que tener en cuenta: la lista vacía.

- Si `sum_all` recibe una lista vacía, deberá devolver `0` .

Hay otro caso más: si recibe una lista de cosas que **no son números**. En ese caso, no hay mucho que hacer, la función `sum_all` no tiene mucho sentido en caso de cosas que no son números. `sum_all` es una función parcila y tendremos que gestionar ese error más adelante, de momento lo dejamos.

Firma y comentarios

- La función irá recorriendo la lista y sumando el valor actual al total hasta entonces.
- Vamos a necesitar una variable para ir almacenda ese total parcial.
- Dicha variable tendrá que ser inicializada a un valor inicial que no altere la suma y que permita que cuando la lista sea la lista vacía, el resultado sea el correcto.

```
def sum_all(numbers):
    # inicializamos la variable total
    # recorremos la lista, sumando el valor actual al total parcial
    # cuando terminamos, devolvemos el total (que ya no será parcial)
```

En una segunda etapa, la función tendría esta pinta:

```
def sum_all(numbers):
    """
    Suma todos los elementos de una lista de numeros.
    Si recibe la lista vacía, devuelve cero
    """
    # inicializamos la variable total
    total = ??
    # recorremos la lista, sumando el valor actual al total parcial
    for number in numbers:
        ???
    # cuando terminamos, devolvemos el total (que ya no será parcial)
    return ??
```

1. Finiquita tú ese patrón con los datos que faltan.
2. Comprueba que funciona tal y como dices, casos especiales incluídos

(2)

Siguiendo el método de arriba, crea la función `multiply_all` , que recibe una lista de números y devuelve el producto de todos ellos.

(3)

Compara las funciones de (1) y (2):

- ¿En qué se parecen?
- ¿En qué se distinguen?
- ¿Ves algún tipo de patrón?

(4)

Crea la función `sum_mult_of_3` :

- Recibe una lista de números
- Devuelve la suma de aquellos que son **múltiplos de 3**

DIVIDE & VENCERÁS

Resuelve primero el subproblema "saber si un número es múltiplo de 3", creando el predicado `is_multiple_3` . Recuerda el operador `%` para obtener el resto de una división.

(5)

Crea la función `count_str` que recibe una lista y devuelve el número de cadenas que hay en ella.

Asegúrate de que funciona cuando:

- solo hay cadenas
- no hay ninguna cadena
- la lista de entrada es la vacía

1. ¿Se parece en algo a `sum_mult_of_3` ?

(6)

Crea la función `max_value` que recibe una lista y devuelve el mayor elemento de la misma.

- ¿Qué crees que debe de hacer si todos son iguales?
- ¿Qué crees que debe de devolver en caso de recibir la lista vacía?

(7)

Es siempre una buena idea que las funciones sean *totales*. A veces lo podemos lograr *ampliando el valor de retorno*. De esta manera, conseguimos eliminar los casos especiales (como que la lista de entrada es la lista vacía). Esto hace que el uso de la función sea más sencillo y se reducen las posibilidades de errores.

> Cerveza para todos. Elimina los casos especiales

¿Cual es el valor máximo de una lista, *cuando la lista está vacía*? En Español, diríamos **ninguno**. En Python, se dice `None`.

`None` es la única instancia del typo `NoneType` y se usa para indicar la nada.

Entre los tipos de Python, juego el mismo papel que el 0 entre los números.

1. Refactoriza tu función `max_value` para que en el caso de que la lista de entrada sea la vacía, que devuelva `None`.
2. Crea la función `min_value` que devuelve el valor mínimo.
3. Compara ambas. ¿Qué partes son comunes y qué partes son distintas?

(8)

Crea la función `concat_all`, que recibe una *lista de cadenas* y devuelve otra cadena, con todas ellas concatenadas. Asegúrate de que funciona correctamente con la lista vacía.

(9)

Compara `concat_all` con `sum_all`. ¿En qué se parecen? ¿En qué se distinguen? ¿Ves algún patrón?

(10)

Crea la función `concat_if_u`. Recibe una lista de cadenas y devuelve la concatenación de todas aquellas que contienen una u.

Recuerda dividir para vencer. ¿Hay algún sub-problema que resolver antes de meter mano al principal?

(11)

Compara `concat_if_u` con `concat_all`. ¿En qué se parecen y en qué se distinguen?

(12)

Compara ahora `sum_mult_of_3` con `concat_if_u`. ¿En qué se parecen y en qué se distinguen?

(13)

Crea la función `all_equal`. Recibe una lista (de lo que sea) y devuelve `True` si *todos los elementos son iguales entre sí* y `False` en caso contrario.

Por ejemplo:

```
all_equal([1,1,1]) -> True  all_equal(["hola", "Mundo"]) -> False
```

1. ¿Qué tipo de función es `all_equal`
2. ¿Qué debería de devolver cuando recibe la lista vacía?
3. ¿Es una función total o parcial?
4. Asegúrate que tu función no recorre elementos innecesarios de la lista

(14)

¿En qué se parece `all_equal` a por ejemplo `sum_mult_of_3` ?

¿En qué se distinguen?

(15)

Crea la función `all_mult_of_3`. Recibe una lista de números y devuelve `True` *si todos son múltiplos de 3*.

1. ¿Qué tipo de función es `all_equal`
2. ¿Qué debería de devolver cuando recibe la lista vacía?
3. ¿Es una función total o parcial?
4. Asegúrate que tu función no recorre elementos innecesarios de la lista

(16)

Crea la función `any_mult_of_3`. Recibe una lista de números y devuelve `True` *si al menos uno es múltiplo de 3*.

1. ¿Qué tipo de función es `any_mult_of_3`
2. ¿Qué debería de devolver cuando recibe la lista vacía?
3. ¿Es una función total o parcial?
4. Asegúrate que tu función no recorre elementos innecesarios de la lista

(17)

Compara `all_mult_of_3 y all_mult_of_3`.

- ¿En qué se parecen?
- ¿En qué se distinguen?

(18)

Crea la función `all_equal_chars`. Recibe una cadena (una secuencia de caracteres) y devuelve `True` si todos los caracteres son iguales. Por ejemplo:

`all_equal_chars('aaaaaaaaaaaa')` devuelve `True` `all_equal_chars('aaaaaaxaaaaaa')` devuelve `False`

- ¿Cual es el caso extremo y qué debe de devolver?
- Función total o parcial?
- ¿Qué tipo de función es?

(19)

Compara `all_equal` con `all_equal_chars`.

¿Cómo lo explicas?

(20)

Crea la función `implode`. Recibe una lista de cadenas y la convierte en una cadena, concatenando todo y respetando el orden.

1. ¿ `implode` es una función reversible?

(21)

Crea la función `is_sum_seq`. Recibe una lista de números y devuelve `True` si cada elemento es la suma de todos los anteriores.

1. La lista tienen que tener al menos 3 elementos (para que el tercero sea la suma de los anteriores). En caso contrario, debe de devolver `False`
2. La función no debe de recorrer más elementos de la lista de los necesarios.
3. El primer elemento que se debe de inspeccionar es el tercero.

Repaso y Conclusiones

NO LEAS ESTO ANTES DE HACER LOS EJERCICIOS

Describe con tus palabras aquello que has visto de común que tienen todas estas funciones.

NO SIGAS ANTES DE HACER ESO

Aquí tienes una pista (https://www.youtube.com/watch?v=q9BtYEnrkg4)

Compresores

Todas estas funciones tienen una cosa en común *reciben una lista de elementos* y devuelven un solo elemento.

Reciben una lista y la **comprimen** a un sólo elemento. Dicho elemento se obtiene *combinando mediante alguna operación* a los elementos de la lista.

Para la operación de compresión, a veces usamos todos los elementos y otras veces solo algunos **seleccionados**.

Para seleccionar, usamos un **predicado**.

Para ir construyendo el valor comprimido, usamos una variable donde vamos *acumulando* los resultados temporales. A esa variable le llamamos el **acumulador**.

Cuando llegamos al final, tenemos el *valor comprimido final* dentro del *acumulador*, y lo devolvemos.

El patrón general de un **compresor** es:

```
def compress(elements):

    # Acumulador con valor inicial
    accum = <initial>

    # iteramos
    for element in elements:
        # combinamos. La operación de combinación
        # puede incluir un predicado que seleccione
        # o no el valor actual para la combinación
        accum = <combine>(element, accum)

    # Tenemos en accum el valor comprimido
    return accum
```

Compresores totales

Podemos forzar a que una función sea total ampliando un poco su tipo de retorno. En concreto, hemos hecho que algunos compresores parciales se convirtiesen en totales al devolver `None` en ciertos casos especiales.

> Cerveza para todos

Esqueleto común y partes específicas

Más arriba hemvos sito el esqueleto común a todo compresor. Lo que es específico de cada uno está entre <>:

- el valor inicial del acumulador

- la operación de combinación (que puede incluir un predicado)

> DRY: Do not repeat yourself

Estamos cometiendo un grave pecado contra el principio DRY y deberíamos bscar una solución.

¿Qué solución se te ocurre a tí?

Más adelante resolveremos esto.

¿Reversible?

1. En tu opinión, ¿los compresores suelen ser funciones *reversibles*?
2. Si tuviesemos muchos casos de input (listas) y output (valor comprimido), ¿crees que algunos compresores serían *irreversibles estimables*?

Destrucción

Ninguna de estas funciones altera o destruye sus parámetros. Por ejemplo, la lista de entrada queda tal cual después de llamar a la función.

> Las funciones que alteran sus parámetros se llaman destructivas. Son más peligrosas que una piraña en un bidé

Nunca implementes funciones destructivas a no ser que sea absolutamente indispensable. Si lo haces, documéntalo muy bien.