

# Introducción a la programación

## Práctica 7: Funciones sobre listas (tipos complejos) - Con algunas propuestas de solución

## 1.3 Suma Total

```
problema suma_total (in s:seq< $\mathbb{Z}$ >) :  $\mathbb{Z}$  {  
  requiere: { True }  
  asegura: { res es la suma de todos los elementos de s }  
}
```

**Nota:** no utilizar la función `sum()` nativa

*Pista:* En este ejercicio estaremos usando una variable que acumula el resultado y luego lo devuelve.

# Solución Suma Total

Versión de Python 3.8 o inferior:

```
from typing import List
```

```
def suma_total(s: List[int]) -> int:
    total: int = 0
    indice_actual: int = 0
    longitud: int = len(s)

    while (indice_actual < longitud):
        valor_actual: int = s[indice_actual]
        total = total + valor_actual
        indice_actual += 1

    return total
```

# Solución Suma Total

Versión de Python 3.9 o superior:

```
def suma_total(s: list [int]) -> int:  
    total: int = 0  
    indice_actual: int = 0  
    longitud: int = len(s)  
  
    while (indice_actual < longitud):  
        valor_actual: int = s[indice_actual]  
        total = total + valor_actual  
        indice_actual += 1  
  
    return total
```

# Debugging

- ▶ Programar es también adquirir habilidades y buenas prácticas, además de poder codificar el problema en un lenguaje de programación específico.
- ▶ En la programación imperativa logramos nuestro objetivo cuando, partiendo de un estado inicial llegamos a un estado final que cumple nuestro propósito. A veces no es simple entender si lo estamos haciendo de forma correcta pues hay muchos estados intermedios.
- ▶ Una habilidad importante para poder comprender esta sucesión de estados es la de poder analizar el código paso a paso.

A esto llamamos *debug*.

# ¿Qué es Debugging y para qué sirve?

1. Podemos ir paso a paso analizando los valores de las variables durante la ejecución
2. Sirve para poder realizar seguimiento del código
3. Podemos avanzar paso a paso o saltar al siguiente breakpoint
4. Podemos terminar la ejecución por la mitad o bien continuar hasta el final
5. Con VSCode podemos agregar breakpoints durante el momento de debugging, o eliminarlos
6. Se pueden agregar breakpoints con condiciones lógicas, por ejemplo: `valor_actual = 7`

# Agregar un breakpoint (punto de detención) en el código

Debemos hacer click a la izquierda del número de línea para agregar el punto de detención en esa línea:

```
1
2  def suma_total(s:[int])-> int:
3      total:int = 0
4      indice_actual:int = 0
5      longitud:int = len(s)
6
7      while (indice_actual < longitud):
8          valor_actual:int = s[indice_actual]
9          total = total + valor_actual
10         indice_actual += 1
11
12     return total
13
```




Figura: Agregamos un breakpoint en la línea 7 del código

# Ejecutar con Debug

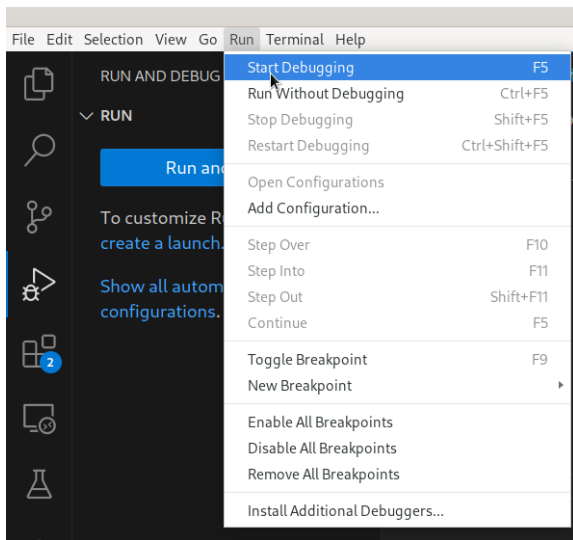
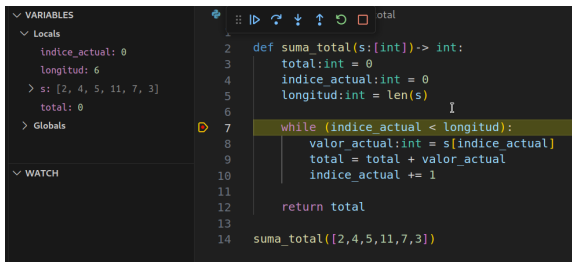


Figura: Ejecutamos el código con la opción Debug



# Usamos los controles de la IDE para desplazarnos



The screenshot shows a Python IDE with a debugger window on the left and a code editor on the right. The debugger window has two tabs: 'VARIABLES' and 'WATCH'. The 'VARIABLES' tab is active, showing the 'Locals' section with the following variables and values:

- `indice_actual`: 0
- `longitud`: 6
- `s`: [2, 4, 5, 11, 7, 3]
- `total`: 0

The 'WATCH' tab is empty. The code editor shows a Python function `suma_total` that calculates the sum of a list `s`. The function is defined as follows:

```
def suma_total(s:[int]) -> int:
    total:int = 0
    indice_actual:int = 0
    longitud:int = len(s)
    while (indice_actual < longitud):
        valor_actual:int = s[indice_actual]
        total = total + valor_actual
        indice_actual += 1
    return total
```

The function is called at line 14: `suma_total([2,4,5,11,7,3])`. The debugger is currently paused at line 7, which is the start of the `while` loop. The line number 7 is highlighted in the left margin of the code editor.

**Figura:** Podemos ver las variables con sus valores al momento del break y usar los controles para movernos

# Usamos los controles de la IDE para desplazarnos



**F5** Continuar hasta el siguiente breakpoint (o si no hay más hasta el final)

**F10** Siguiente paso saltando ingresar a la función que se esté evaluando en esta línea

**F11** Siguiente paso ingresando a la función que se esté evaluando en esa línea

**Shift+F11** Salir de la evaluación de la función a la que se ingresó

**Ctrl + Shift + F5** Reiniciar el debug desde el principio

**Shift + F5** Detener el debugging

## Solución Suma Total - Otras soluciones

```
def suma_total2(s:list[int])-> int:
    total:int = 0
    longitud:int = len(s)
    for ind in range(0,longitud):
        total = total + s[ind]
    return total
```

```
def suma_total3(s:list[int])-> int:
    total:int = 0
    for valor in s:
        total = total + valor
    return total
```

## Ejercicio 1.1: Pertenece

```
problema pertenece ( in s:seq< $\mathbb{Z}$ >, in e:  $\mathbb{Z}$ ) : Bool {  
  requiere: { True }  
  asegura: { (res = true)  $\leftrightarrow$  e  $\in$  s }  
}
```

Implementar al menos de 3 formas distintas éste problema.

# Soluciones Pertenece

```
def pertenece_1(s: list[int], e: int) -> bool:  
    longitud: int = len(s)  
    indice_actual: int = 0  
    pertenece: bool = False  
  
    while (indice_actual < longitud):  
        if (s[indice_actual] == e):  
            pertenece = True  
            indice_actual = indice_actual + 1  
  
    return pertenece
```

## Soluciones Pertenece

```
def pertenece_2(s:[int], e:int) -> bool:  
    longitud:int = len(s)  
    indice_actual:int = 0  
    pertenece:bool = False  
  
    while ((indice_actual < longitud) and  
        (not pertenece)):  
        if (s[indice_actual] == e):  
            pertenece = True  
        indice_actual = indice_actual + 1  
  
    return pertenece
```

# Soluciones Pertenece

```
def pertenece_2(s:[int], e:int) -> bool:  
    return e in s  
# OJO, es probable que en el parcial  
# no se pueda usar "in".
```

## Ejercicio 1.7

7. Analizar la fortaleza de una contraseña. El parámetro de entrada será un *string* con la contraseña, y la salida otro *string* con tres posibles valores: VERDE, AMARILLA y ROJA.



## Ejercicio 1.7: fortaleza de una contraseña

► **VERDE** si:

- a) longitud es mayor a 8 caracteres
- b) tiene al menos una minúscula
- c) tiene al menos una mayúscula
- d) tiene al menos un número (0...9)

## Ejercicio 1.7: fortaleza de una contraseña

► **VERDE** si:

- a) longitud es mayor a 8 caracteres
- b) tiene al menos una minúscula
- c) tiene al menos una mayúscula
- d) tiene al menos un número (0...9)

► **ROJA** si:

- a) longitud es menor a 5 caracteres

## Ejercicio 1.7: fortaleza de una contraseña

- ▶ **VERDE** si:
  - a) longitud es mayor a 8 caracteres
  - b) tiene al menos una minúscula
  - c) tiene al menos una mayúscula
  - d) tiene al menos un número (0...9)
- ▶ **ROJA** si:
  - a) longitud es menor a 5 caracteres
- ▶ **AMARILLA** en caso contrario

## Ejercicio 1.7: el código ASCII

- ▶ Asocia un número a cada carácter

## Ejercicio 1.7: el código ASCII

- ▶ Asocia un número a cada carácter
- ▶ Determina un orden entre los caracteres

## Ejercicio 1.7: el código ASCII

- ▶ Asocia un número a cada carácter
- ▶ Determina un orden entre los caracteres

47	/	64	@	96	'
48	<b>0</b>	65	<b>A</b>	97	<b>a</b>
49	<b>1</b>	66	<b>B</b>	98	<b>b</b>
50	<b>2</b>	67	<b>C</b>	99	<b>c</b>
51	<b>3</b>	68	<b>D</b>	100	<b>d</b>
52	<b>4</b>	69	<b>E</b>	101	<b>e</b>
...	...	...	...	...	...
57	<b>9</b>	90	<b>Z</b>	122	<b>z</b>

Cuadro: parte de la codificación ASCII

**Nota:** se puede ver la tabla completa en [elcodigoascii.com.ar](http://elcodigoascii.com.ar)

## Ejercicio 1.7: el código ASCII

- ▶ Python permite realizar comparaciones entre caracteres basadas en el orden dado por el código ASCII
- ▶ **Probar en Python imprimir las siguientes expresiones bool:**
  - ▶ `print('a' < 'b')`
  - ▶ `print('A' > 'Z')`
  - ▶ `print('0' < '5')`
  - ▶ `print('@' < 'E')`

## Ejercicio 1.7: Cómo establecer condiciones sobre los caracteres

- ▶ **¿Cómo puedo saber si un carácter es un número?**
- ▶  $\text{es\_un\_numero} = (\text{carácter} \leq '9') \text{ and } (\text{carácter} \geq '0')$
- ▶ Podemos usar `es_un_numero` como una **condición** para analizar cuando vamos recorriendo la contraseña.
- ▶ El mismo análisis haremos para cualquier otra condición. A continuación veremos opciones de algoritmos para cualquier condición.



## Ejercicio 1.7: dos formas de verificar “tiene al menos un”

```
i: int = 0
vale_condición: bool = False
while i < len(contrasena):
    if condición:
        vale_condición: bool = True
    i += 1
```

## Ejercicio 1.7: dos formas de verificar “tiene al menos un”

```
i: int = 0
vale_condición: bool = False
while i < len(contrasena) and not condición:
    i += 1
vale_condición: bool = i < len(contrasena)
```

## Ejercicio 1.7: comparativa “tiene al menos un número”

**contrasena:** `str` = “cl4v3”

<b>i</b>	<b>contrasena[i]</b>	<b>Algoritmo 1</b>	<b>Algoritmo 2</b>
0	c	vale_condición = False	vale_condición = False

## Ejercicio 1.7: comparativa “tiene al menos un número”

**contrasena:** `str` = “cl4v3”

<b>i</b>	<b>contrasena[i]</b>	<b>Algoritmo 1</b>	<b>Algoritmo 2</b>
0	c	vale_condición = False	vale_condición = False
1	l	vale_condición = False	vale_condición = False

## Ejercicio 1.7: comparativa “tiene al menos un número”

**contrasena:** `str` = “cl4v3”

i	contrasena[i]	Algoritmo 1	Algoritmo 2
0	c	vale_condición = False	vale_condición = False
1	l	vale_condición = False	vale_condición = False
2	4	vale_condición = True	-

## Ejercicio 1.7: comparativa “tiene al menos un número”

**contrasena:** `str` = “cl4v3”

i	contrasena[i]	Algoritmo 1	Algoritmo 2
0	c	vale_condición = False	vale_condición = False
1	l	vale_condición = False	vale_condición = False
2	4	vale_condición = True	-
3	v	vale_condición = True	-

## Ejercicio 1.7: comparativa “tiene al menos un número”

**contrasena:** `str` = “cl4v3”

i	contrasena[i]	Algoritmo 1	Algoritmo 2
0	c	vale_condición = False	vale_condición = False
1	l	vale_condición = False	vale_condición = False
2	4	vale_condición = True	-
3	v	vale_condición = True	-
4	3	vale_condición = True	-

## Ejercicio 1.7: comparativa “tiene al menos un número”

**contrasena:** `str` = “cl4v3”

i	contrasena[i]	Algoritmo 1	Algoritmo 2
0	c	vale_condición = False	vale_condición = False
1	l	vale_condición = False	vale_condición = False
2	4	vale_condición = True	-
3	v	vale_condición = True	-
4	3	vale_condición = True	-
5	-	vale_condición = True	vale_condición = $i < \text{len}(\text{contrasena}) =$ True

**Cuadro:** seguimiento paso a paso de ambos algoritmos



## Solución Ejercicio 1.7

```
def fortaleza_contrasena(contrasena: str) -> str:  
  
    # Verifico longitud mayor a 8  
    longitud_mayor_a_8: bool = len(contrasena) > 8  
  
    # Verifico longitud menor a 5  
    longitud_menor_a_5: bool = len(contrasena) < 5
```

## Solución Ejercicio 1.7

```
# Verifico tiene una mayuscula
```

```
i: int = 0
```

```
tiene_mayus: bool = False
```

```
while i < len(contrasena):
```

```
    if contrasena[i] >= 'A' and contrasena[i] <= 'Z':  
        tiene_mayus = True
```

```
    i += 1
```

```
# Verifico tiene una minuscula
```

```
tiene_minuscula: bool = False
```

```
for i in range(0, len(contrasena)):
```

```
    if contrasena[i] >= 'a' and contrasena[i] <= 'z':  
        tiene_minuscula = True
```

## Solución Ejercicio 1.7

*# Verifico tiene un dígito numerico*

```
i: int = 0
```

```
while i < len(contrasena) and
```

```
    not(contrasena[i] >= '0' and contrasena[i] <= '9')  
    i += 1
```

```
tiene_num: bool = i < len(contrasena)
```

*# Devuelvo nivel de fortaleza segun las condiciones*

```
if longitud_mayor_a_8 and tiene_mayus and  
    tiene_minuscula and tiene_num:
```

```
    return 'VERDE'
```

```
elif longitud_menor_a_5:
```

```
    return 'ROJA'
```

```
else:
```

```
    return 'AMARILLA'
```

## Ejercicio 2.1

Dada una lista de números, en las posiciones pares borra el valor original y coloca un cero. Esta función modifica el parámetro ingresado, es decir, la lista es un parámetro de tipo inout.

## Soluciones Ejercicio 2.1

```
def es_par(num: int) -> bool:  
    return (num % 2 == 0)
```

```
def reemplazar_pos_pares_por_cero(s: list [int]) -> Non  
    indice_actual: int = 0  
    longitud: int = len(s)  
    while (indice_actual < longitud):  
        if (es_par(indice_actual)):  
            s[indice_actual] = 0  
        indice_actual += 1
```

```
def reemplazar_pos_pares_por_cero2(s: list [int]) -> No  
    for ind in range(0, len(s), 2):  
        s[ind] = 0
```

## 5.2 Pertenece a Cada Uno

```
problema pertenece_a_cada_uno_version_2 (in s:seq<seq<ℤ>>,
in e:ℤ, out res: seq<Bool>) {
  requiere: { True }
  asegura: {|res| = |s|}
  asegura: { Para todo  $i \in \mathbb{Z}$  si
     $0 \leq i < |s| \rightarrow (res[i] = true \leftrightarrow pertenece(s[i], e))$  }
}
```

**Nota:** Reutilizar la función `pertenece()` implementada previamente para listas

# Soluciones Pertenece a cada uno

```
def pertenece_a_cada_uno_version_2  
(s: list[list[int]], e: int, res: list[bool]) -> None:
```

```
    indice_actual: int = 0  
    longitud: int = len(s)  
    res.clear()
```

```
    while(indice_actual < longitud):  
        lista_actual: int = s[indice_actual]  
        res.append(pertenece(lista_actual, e))  
        indice_actual += 1
```

```
def pertenece_a_cada_uno_version_2  
(s: list[list[int]], e: int, res: list[bool]) -> None:  
    res.clear()  
    for i in range(len(s)):  
        res.append(pertenece(s[i], e))
```