

Introducción a la programación

Debugging

Contenido de la clase de hoy:

1. Uso de Debugger en VSCode

Este código no anda bien, ¿Cómo sabemos que le pasa?

Queremos ver que pasa para `[2, 3, 6, 9, 2]`

```
1  def contar_pares_impares(numeros: List[int]) -> tuple[
    int, int]:
2      pares_mayores_a_5: int = 0
3      pares_menores_a_5: int = 0
4      impares_mayores_a_4: int = 0
5      impares_menores_a_4: int = 0
6      for n in numeros:
7          if n%2 == 0:
8              if n>5 :
9                  pares_mayores_a_5 +=1
10                 else:
11                     pares_menores_a_5 +=1
12             else:
13                 if n>5 :
14                     impares_mayores_a_4 +=1
15                 else:
16                     pares_menores_a_5 +=1
17     return pares_menores_a_5 + pares_mayores_a_5 ,
        impares_mayores_a_4 + impares_menores_a_4
```

Resultado incorrecto de contar_pares_impares

Entrada:

2



par

3



impar

6



par

9



impar

2



par

Esperado: (3, 2) (3 pares, 2 impares)

Obtenido: (4, 1)

Idea 1: Agrego prints

```
1 def contar_pares_impares_con_prints(numeros : List[int]) -> tuple[int,int]:
2     pares_mayores_a_5: int = 0
3     pares_menores_a_5: int = 0
4     impares_mayores_a_4: int = 0
5     impares_menores_a_4: int = 0
6
7     for n in numeros:
8         @print(f"Analizando numero: {n}")@
9         if n % 2 == 0:
10             print("Es par")
11             if n > 5:
12                 pares_mayores_a_5 += 1
13                 print("Mayor a 5")
14             else:
15                 pares_menores_a_5 += 1
16                 print("Menor o igual a 5")
17         else:
18             print("Es impar")
19             if n > 5:
20                 impares_mayores_a_4 += 1
21                 print("Mayor a 5")
22             else:
23                 pares_menores_a_5 += 1
24                 print("Menor o igual a 5")
25
26         print(f"pares_menores_a_5 = {pares_menores_a_5}")
27         print(f"pares_mayores_a_5 = {pares_mayores_a_5}")
28         print(f"impares_menores_a_4 = {impares_menores_a_4}")
29         print(f"impares_mayores_a_4 = {impares_mayores_a_4}")
30
31     return pares_menores_a_5 + pares_mayores_a_5, impares_mayores_a_4 +
        impares_menores_a_4
```

**YO DEBUGGEANDO
CON PRINTS**

LOS BUGS



imgflip.com

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.

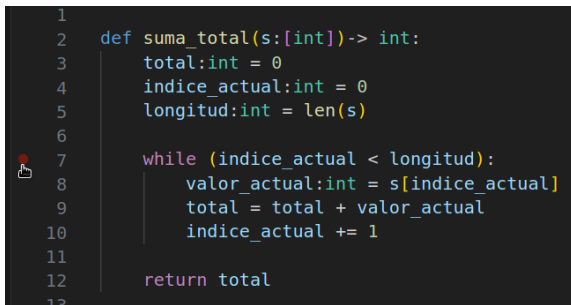
A esto llamamos *debug*.

¿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
```

The image shows a code editor with a dark background. On the left side, there is a vertical line of numbers from 1 to 13. A red dot, representing a breakpoint, is placed to the left of the number 7. A small mouse cursor icon is positioned over this red dot. The code on the right is a Python function named 'suma_total' that takes a list 's' and returns the sum of its elements. The function uses a 'while' loop to iterate through the list, adding each element to a 'total' variable. The code is color-coded: keywords are in blue, variables and function names are in green, and numbers are in yellow.

Figura: Agregamos un breakpoint en la línea 7 del código

Ejecutar con Debug

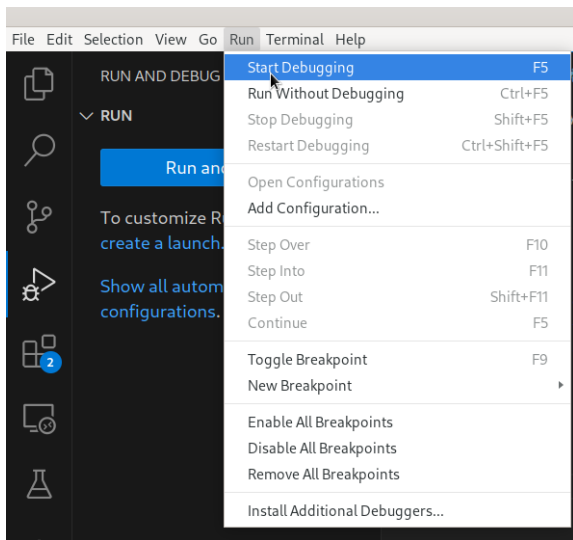
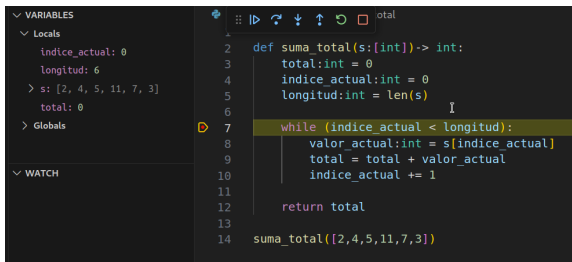


Figura: Ejecutamos el código con la opción Debug

Usamos los controles de la IDE para desplazarnos



The image shows a Python IDE interface. On the left, there is a 'VARIABLES' panel with a 'Locals' section containing the following variables and values:

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

Below this is a 'WATCH' section which is currently empty. The main editor area shows a Python function `suma_total` with the following code:

```
1 def suma_total(s:[int])-> int:
2     total:int = 0
3     indice_actual:int = 0
4     longitud:int = len(s)
5
6     while (indice_actual < longitud):
7         valor_actual:int = s[indice_actual]
8         total = total + valor_actual
9         indice_actual += 1
10
11     return total
12
13 suma_total([2,4,5,11,7,3])
14
```

The line `while (indice_actual < longitud):` is highlighted in green, indicating the current execution point. The cursor is positioned at the end of this line. The 'total' variable is also visible in the top right corner of the editor area.

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

Ahora usemos el debugger con el código anterior

```
1  def contar_pares_impares(numeros: List[int]) -> tuple[
    int, int]:
2      pares_mayores_a_5: int = 0
3      pares_menores_a_5: int = 0
4      impares_mayores_a_4: int = 0
5      impares_menores_a_4: int = 0
6      for n in numeros:
7          if n%2 == 0:
8              if n>5 :
9                  pares_mayores_a_5 +=1
10             else:
11                 pares_menores_a_5 +=1
12         else:
13             if n>5 :
14                 impares_mayores_a_4 +=1
15             else:
16                 pares_menores_a_5 +=1
17     return pares_menores_a_5 + pares_mayores_a_5 ,
        impares_mayores_a_4 + impares_menores_a_4
```

Fibonacci no recursivo

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

```
problema fibonacci (n: ℤ) : ℤ {  
  requiere: { n ≥ 0 }  
  asegura: { resultado = fib(n) }  
}
```

Fibonacci no recursivo

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

problema fibonacci ($n: \mathbb{Z}$) : \mathbb{Z} {
 requiere: { $n \geq 0$ }
 asegura: { *resultado* = $fib(n)$ }
}

Implementemos, probemos y debuggeemos este código:

```
1 def fibonacci_no_recursivo (n: int) -> int:
2     if n <= 1:
3         return n
4     un_fibo: int = 0
5     un_fibo_sig: int = 1
6     i: int = 2
7     while i <= n:
8         aux: int = un_fibo + un_fibo_sig
9         un_fibo = un_fibo_sig
10        un_fibo_sig = aux
11        i = i+1
12    return un_fibo
```

Buscando primos...

Implementemos, probemos y debuggeemos este código:

```
1 def es_primo (n: int) -> bool:
2     cant_divisores: int = 0
3     i: int = 1
4     while i < n and cant_divisores < 2:
5         if n % i == 0:
6             cant_divisores += 1
7             i += 1
8
9     return cant_divisores < 2 and i == n
```


Buscando primos...

Implementemos, probemos y debuggeemos este código:

```
1 def buscar_nesimo_primo(n: int) -> int:  
2     cant_primos: int = 0  
3     i: int = 2  
4     while cant_primos < n:  
5         if es_primo(i):  
6             cant_primos += 1  
7  
8     return i
```

A programar! suma_matriz_fila_col

Supongamos que tenemos una matriz M donde para cada posición (i,j) vale que $M_{ij} = i + j$. Esa matriz se ve así...

$$\begin{bmatrix} 1+1 & 1+2 & 1+3 & \dots & 1+c \\ 2+1 & 2+2 & 2+3 & \dots & 2+c \\ 3+1 & 3+2 & 3+3 & \dots & 3+c \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ f+1 & f+2 & f+3 & \dots & f+c \end{bmatrix}$$

A programar! suma_matriz_fila_col

Supongamos que tenemos una matriz M donde para cada posición (i,j) vale que $M_{ij} = i + j$. Esa matriz se ve así...

$$\begin{bmatrix} 1+1 & 1+2 & 1+3 & \dots & 1+c \\ 2+1 & 2+2 & 2+3 & \dots & 2+c \\ 3+1 & 3+2 & 3+3 & \dots & 3+c \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ f+1 & f+2 & f+3 & \dots & f+c \end{bmatrix}$$

Queremos obtener, dados f y c que indican la cantidad de filas y la cantidad de columnas de la matriz, cuál es el resultado de sumar todos los elementos de la matrix. Requiere: $f \geq 1 \wedge c \geq 1$

A programar! `piramide_de_numeros`

Implementar un programa en Python que, dado un número $n > 0$, imprima una *pirámide de números*. Por ejemplo, para $n = 4$, debería imprimir:

```
1          1
2        121
3      12321
4    1234321
```