

## +TADs & Debugging

Introducción a la Programación

1

## IP - AED I: Temario de la clase

- ▶ Debugging
  - ▶ Repaso del concepto de transformación de estados y ejecución simbólica
  - ▶ ¿Qué es el DEBUG? ¿Para qué sirve?
  - ▶ Debugging en VSCode
- ▶ Continuamos con TADs
  - ▶ Repaso: ¿Qué es un TAD?
  - ▶ TAD Pila utilizando LifoQueue
  - ▶ TAD Cola utilizando Queue
  - ▶ TAD Diccionario
  - ▶ Manejo de archivos

2

## Transformación de estados

Repasando

- ▶ Llamamos **estado** de un programa a los valores de todas sus variables en un punto de su ejecución:
  - ▶ Antes de ejecutar la primera instrucción.
  - ▶ Entre dos instrucciones.
  - ▶ Después de ejecutar la última instrucción.
- ▶ Veremos la ejecución de un programa como una **sucesión de estados**.
- ▶ La asignación es la instrucción que transforma estados.
- ▶ El resto de las instrucciones son de control: modifican el flujo de ejecución es decir, el orden de ejecución de las instrucciones.

3

## Ejemplo de *transformación de estados*

Repasando

```
def ejemplo() -> int:
    x: int = 0
    x = x + 3
    x = 2 * x
    return x
```

Ejemplo de transformación de estados:

```
x = 0
//Estado 1 x == 0
x = x + 3
//Estado 2 x == 3
x = 2 * x
//Estado 3 x == 6
```

4

## Ejecución simbólica

Repasando

```
def suc(x: int) -> int:
    //estado a;
    x = x + 2
    //estado b
    //vale x == x@a+2;
    «En el estado b, x vale lo que valía en el estado a más 2»
    x = x - 1
    //estado c
    //vale x == x@b-1;
    «En el estado c, x vale lo que valía en el estado b menos 1»
    return x
```

- ▶ De esta manera, mediante la transformación de estados, podremos realizar una ejecución simbólica del programa, declarando cuánto vale cada variable, en cada estado del programa, en función de los valores anteriores.
- ▶ Algunas técnicas de verificación estática utilizan estos recursos.

5

## Debugging

- ▶ La mayoría de los IDEs nos brindan una herramienta MUY poderosa llamada **DEBUG**
- ▶ Con esta herramienta vamos a poder ir siguiendo la ejecución del programa *paso a paso*
- ▶ Esta herramienta nos permite:
  - ▶ ir siguiendo el flujo de ejecución de las instrucciones
  - ▶ ir visualizando como las instrucciones (asignaciones) del programa van transformando los estados
- ▶ La ejecución simbólica se vuelve real y podemos ver la evolución concreta de cada variable
- ▶ Es una herramienta fundamental para encontrar errores (**BUGS**) en nuestro código

6

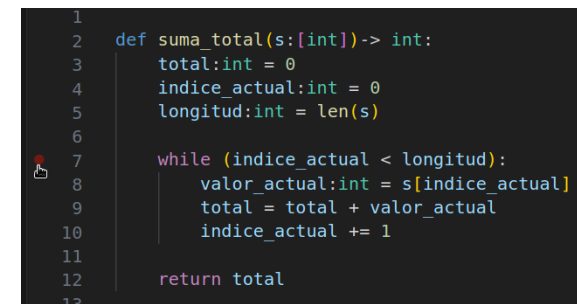
## ¿Qué es Debugging y para qué sirve?

1. Podemos ir paso a paso analizando los valores de las variables durante la ejecución (antes y después de cada instrucción)
2. Sirve para poder realizar seguimiento del código y encontrar errores
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`

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

8

## Ejecutar con Debug

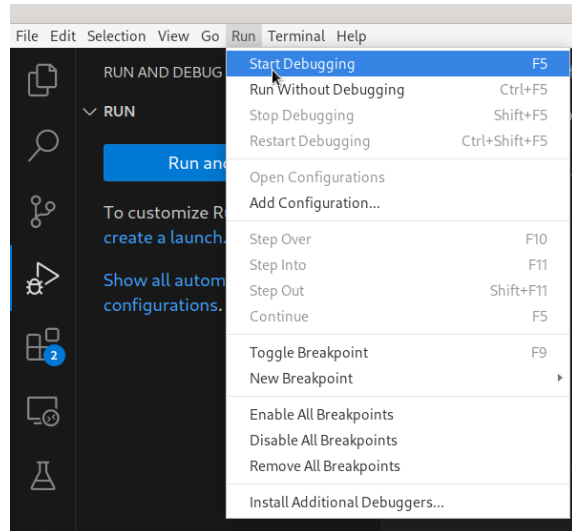


Figura: Ejecutamos el código con la opción Debug

9

## Usamos los controles de la IDE para desplazarnos

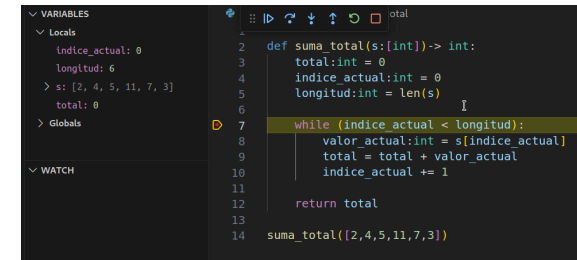


Figura: Podemos ver las variables con sus valores al momento del break y usar los controles para movernos

10

## Usamos los controles de la IDE para desplazarnos



- F5** Continuar hasta el siguiente breakpoint (o si no hay más hasta el final)
- F10** Siguiendo paso saltando ingresar a la función que se esté evaluando en esta línea
- F11** Siguiendo 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

11

## Tipos Abstractos de Datos

### Repasando

Un Tipo Abstracto de Datos (TAD) es un modelo que define valores y las operaciones que se pueden realizar sobre ellos.

- Se denomina abstracto ya que la intención es que quien lo utiliza, no necesita conocer los detalles de la representación interna o bien el cómo están implementadas sus operaciones.

El tipo lista que estuvimos viendo es un TAD:

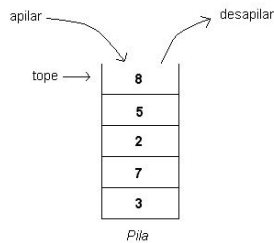
- Se define como una serie de elementos consecutivos
- Tiene diferentes operaciones asociadas: append, remove, etc
- Desconocemos cómo se usa/guarda la información almacenada dentro del tipo

12

## Pila

Una pila es una lista de elementos de la cual se puede extraer el último elemento insertado.

- ▶ También se conocen como listas LIFO (Last In - First Out / el último que entra es el primero que sale)
- ▶ Operaciones básicas
  - ▶ apilar: ingresa un elemento a la pila
  - ▶ desapilar: saca el último elemento insertado
  - ▶ tope: devuelve (sin sacar) el último elemento insertado
  - ▶ vacía: retorna verdadero si está vacía



13

## Pila

- ▶ En Python, el tipo lista provee los métodos necesarios para poder usar una lista como una pila
- ▶ También, podemos importar el tipo LifoQueue del módulo queue, que nos da una implementación de Pila

```
from queue import LifoQueue  
pila = LifoQueue()
```

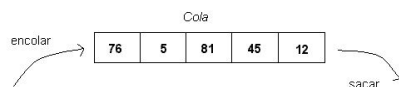
- ▶ Operaciones implementadas en el tipo:
  - ▶ apilar: ingresa un elemento a la cola
    - ▶ `put`
  - ▶ desapilar: devuelve y quita el último elemento insertado
    - ▶ `get`
  - ▶ tope: devuelve (sin sacar) el último elemento insertado
    - ▶ **No está implementado**
  - ▶ vacía: retorna verdadero si está vacía
    - ▶ `empty`

14

## Cola

Una cola es una lista de elementos en donde siempre se insertan nuevos elementos al final de la lista y se extraen elementos desde el inicio de la lista.

- ▶ También se conocen como listas FIFO (First In - First Out / el primero que entra es el primero que sale)
- ▶ Operaciones básicas
  - ▶ encolar: ingresa un elemento a la cola
  - ▶ sacar: saca el primer elemento insertado
  - ▶ vacía: retorna verdadero si está vacía



15

## Cola

- ▶ En Python, el tipo lista provee los métodos necesarios para poder usar una lista como una cola
- ▶ También, podemos importar el tipo Queue del módulo queue, que nos da una implementación de Cola

```
from queue import Queue  
cola = Queue()
```

- ▶ Operaciones implementadas en el tipo:
  - ▶ encolar: ingresa un elemento a la cola
    - ▶ `put`
  - ▶ desencolar: saca el primer elemento insertado
    - ▶ `get`
  - ▶ vacía: retorna verdadero si está vacía
    - ▶ `empty`

16

## Diccionario

Un diccionario es una estructura de datos que permite almacenar y organizar pares clave-valor.

- ▶ Las claves deben ser inmutables (como cadenas de texto, números, etc), mientras que los valores pueden ser de cualquier tipo de dato.
- ▶ La clave actúa como un identificador único para acceder a su valor correspondiente.
- ▶ Los diccionarios son mutables, lo que significa que se pueden modificar agregando, eliminando o actualizando elementos.
- ▶ No ordenados: Los elementos dentro de un diccionario no tienen un orden específico. No se garantiza que se mantenga el orden de inserción de los elementos.

```
diccionario = { clave1:valor1, clave2:valor2, clave3:valor3 }
```

- ▶ Operaciones básicas de un diccionario:
  - ▶ Agregar un nuevo par Clave-Valor
  - ▶ Eliminar un elemento
  - ▶ Modificar el valor de un elemento
  - ▶ Verificar si existe una clave guardada
  - ▶ Obtener todas las claves
  - ▶ Obtener todos los elementos

17

## Diccionario

Un diccionario es una estructura de datos que permite almacenar y organizar pares clave-valor.

- ▶ El valor puede ser cualquier tipo de dato, en particular podría ser otro diccionario

```
infoPaisFrancia = {'Capital':'París',  
                  'Campeonatos de Mundo':2}  
  
infoPaisArgentina = {'Capital':'Buenos Aires',  
                     'Campeonatos de Mundo':3}  
  
infoPaisChile = {'Capital':'Santiago',  
                'Campeonatos de Mundo':0}  
  
infoPaises = {'Chile': infoPaisChile ,  
              'Argentina': infoPaisArgentina,  
              'Francia':infoPaisFrancia}
```

18

## Manejo de Archivos

El manejo de archivos, también puede pensarse mediante la abstracción que nos brindan los TADs

- ▶ Necesitamos una operación que nos permita abrir un archivo
- ▶ Necesitamos una operación que nos permita leer sus líneas
- ▶ Necesitamos una operación que nos permita cerrar un archivo

```
# Abrir un archivo en modo lectura  
archivo = open("archivo.txt", "r")
```

```
# Leer el contenido del archivo  
contenido = archivo.read()  
print(contenido)
```

```
# Cerrar el archivo  
archivo.close()
```

19

## Manejo de Archivos

```
archivo = open("PATH AL ARCHIVO", MODO, ENCODING)
```

- ▶ Algunos de los modos posibles son: escritura (w), lectura (r), texto (t - es el default)
- ▶ El encoding se refiere a como está codificado el archivo: UTF-8 o ASCII son los más frecuentes.

Operaciones básicas

- ▶ Lectura de contenido:
  - ▶ read(size): Lee y devuelve una cantidad específica de caracteres o bytes del archivo. Si no se especifica el tamaño, se lee el contenido completo.
  - ▶ readline(): Lee y devuelve la siguiente línea del archivo.
  - ▶ readlines(): Lee todas las líneas del archivo y las devuelve como una lista.
- ▶ Escritura de contenido:
  - ▶ write(texto): Escribe un texto en el archivo en la posición actual del puntero. Si el archivo ya contiene contenido, se sobrescribe.
  - ▶ writelines(líneas): Escribe una lista de líneas en el archivo. Cada línea debe terminar con un salto de línea explícito.

20

## ¿Podremos implementar este problema?

```
problema invertirTexto(in archivoOrigen: string, in archivoDestino:  
string) : {  
  requiere: {El archivo archivoOrigen debe existir.}  
  asegura: {Se crea un archivo llamado archivoDestino cuyo contenido  
será el resultado de hacer un reverse en cada una de sus filas en orden  
invertido}  
  asegura: {Si el archivo archivoDestino existia, se borrará todo su  
contenido anterior}  
}
```