Algoritmos y Estructuras de Datos III

Profesor:
Rómulo Arceri

Tecnicatura Superior en Análisis de Sistemas

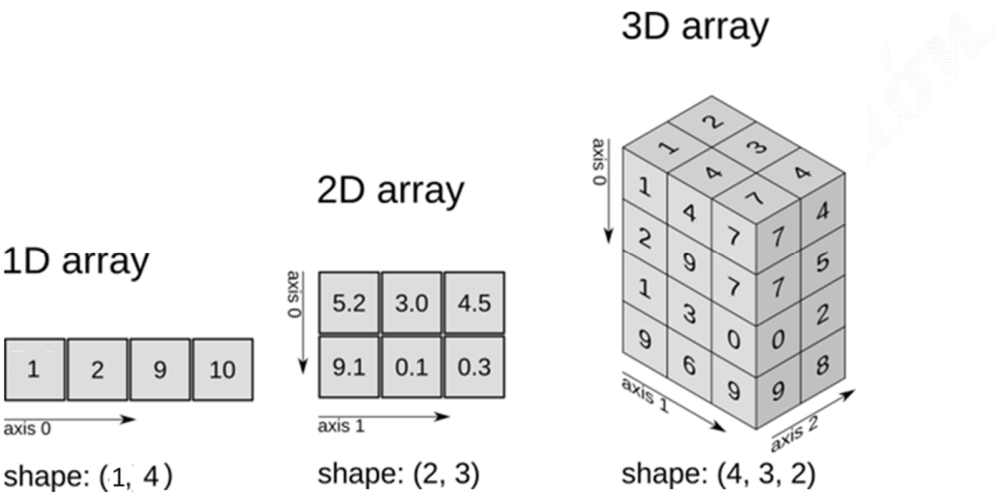
Clase 1

Contenido

Arrays. Tipos vector, 2d y 3d 3
Arrays. Implementación 3
Arrays. import 4
Crear un array vacío, Modificar y agregar elementos 5
Arrays. Eliminar elementos 6
Funciones 7
¿Qué es una función? ¿Cómo la ejecutó? 7
Funciones. Devolución de valores (Return). 7
Funciones. Parámetros y argumentos 8
Funciones. Alcance de las variables 9
Parámetros y argumentos - Ejercicios 10
Funciones Recursivas 11
¿Qué es la recursividad? 11
¿Por qué utilizar la recursividad? 12
Resolución de un problema sin utilizar funciones recursivas 12
Resolución de un problema sin utilizar funciones recursivas 13
Las tres leyes de la recursividad 14
Depurando la función 15
Cuenta regresiva hasta cero a partir de un número 17

Arrays. Tipos vector, 2d y 3d

Los arrays son una zona de almacenamiento continuo, que incluye una serie de elementos del mismo tipo. Desde el punto de vista lógico un arreglo (arrays) tipo vector se puede ver como un conjunto de elementos ordenados en fila.



El dimensionamiento de un arreglo define el tipo de estructura sobre la que trabajaremos. La misma está definida por la cantidad de columnas (podríamos llamarlo eje X) y la cantidad de filas (eje Y).

Para una lista de valores se crea un array de una dimensión, también conocido como **vector 1D**

Para una lista de listas de valores se crea un array de dos dimensiones, también conocido como **matriz 2D**.

Para una lista de listas de listas de valores se crea un array de tres dimensiones, también conocido como **cubo 3D**.

Y así sucesivamente. No hay límite en el número de dimensiones del array más allá de la memoria disponible en el sistema (nD)

Arrays. Implementación

Los arrays en python se pueden implementar

- Mediante la utilización de estructuras de datos :
  - Listas, Cadenas, Diccionarios, Conjuntos, Tuplas.

```
vector_lista = [1,2,9,10]

array2d_lista = [ [5.2,3.0,4.5]  , [9.1,0.1,0.3] ]

array3d_lista = [ [ [1,2],[4,3],[7,4] ] , [ [2,11],[9,6],[7,5] ] , [ [1,15],[3,19],[0,2] ] , [ [9,20],[6,14],[9,8] ] ]
```

- Mediante la importación de la librería array
  - Permite crear vectores o matrices de 1 dimensión, de un solo tipo de datos que pueden ser enteros o flotantes.

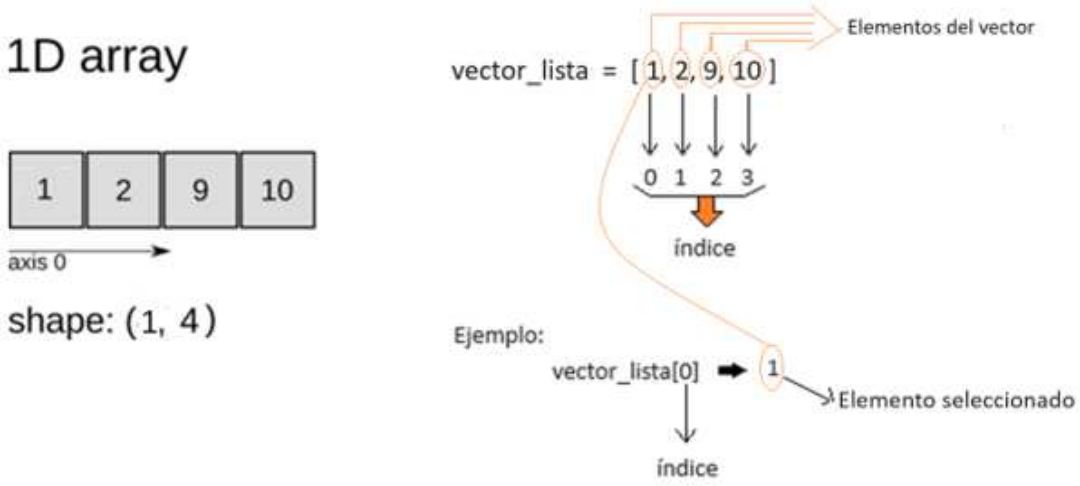
```
import array as arr
num = arr.array('i', [1, 2, 3, 4, 5, 6])
```

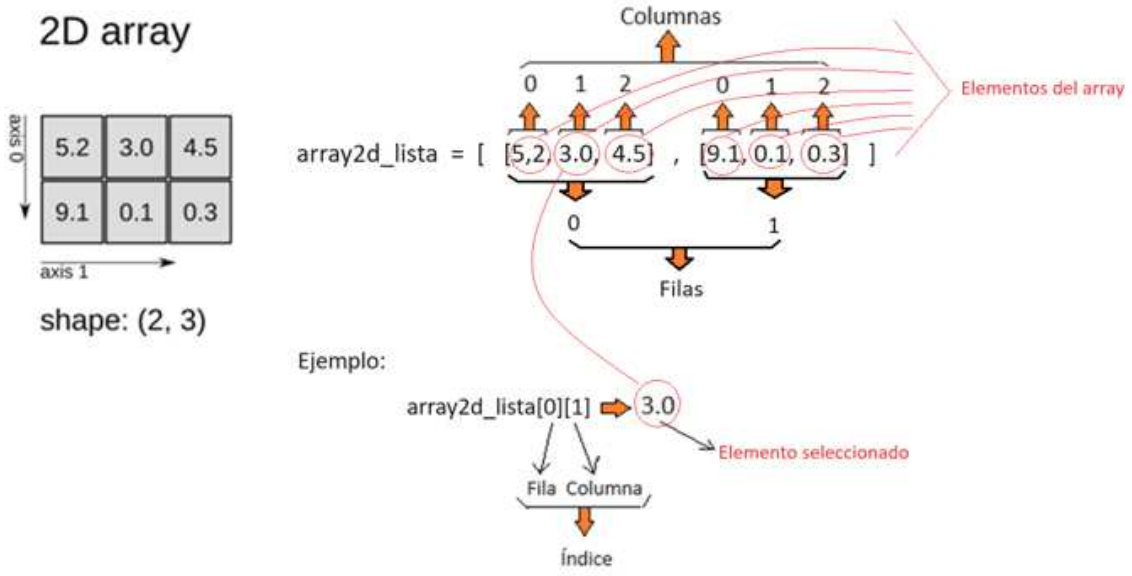
- Mediante la importación de la librería numpy
  - Esta es una biblioteca para Python que da soporte para crear vectores y matrices grandes multidimensionales, junto con una gran colección de funciones matemáticas de alto nivel para operar con ellas. (Wikipedia)

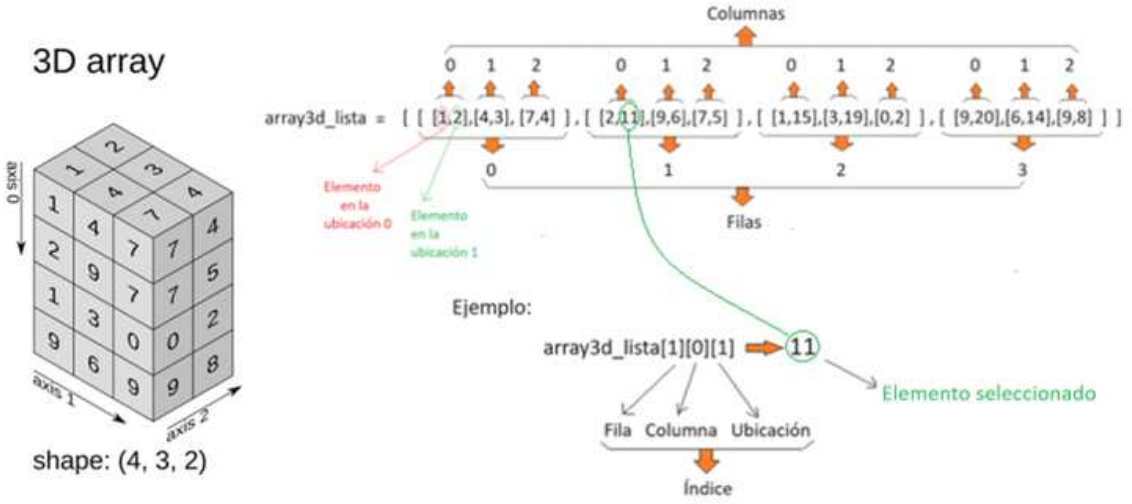
```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
b = np.array([[1, 2, 3], [4, 5, 6]])
c = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

Análisis:

- Un elemento individual dentro de un arreglo es accedido por el uso de un índice.
- Un índice describe la posición de un elemento dentro de un arreglo.
- En Python el primer elemento tiene el índice cero.



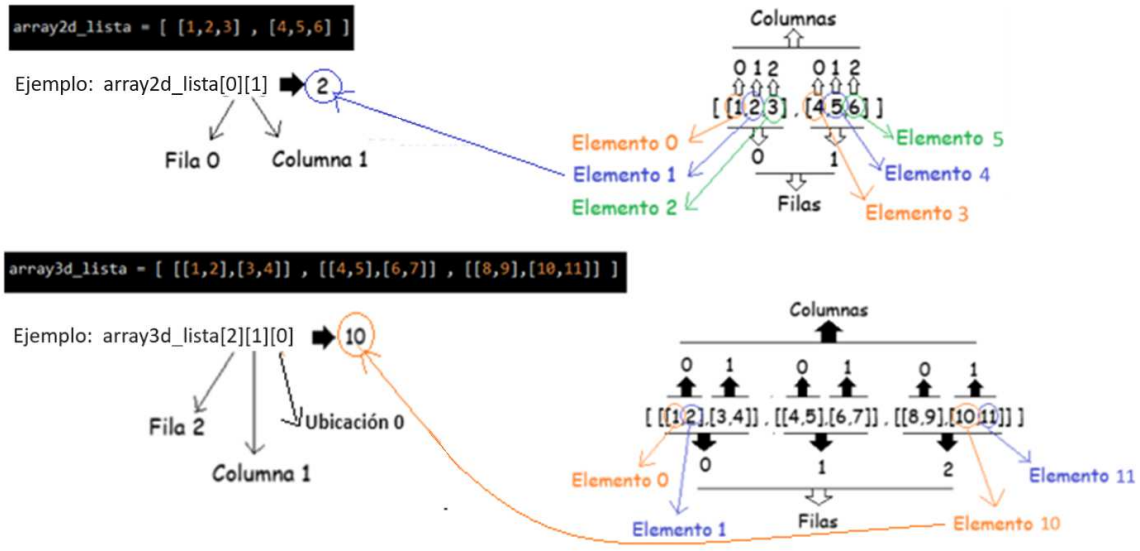





```
vector_lista = [1,2,9,10]

array2d_lista = [ [5.2,3.0,4.5]  , [9.1,0.1,0.3] ]

array3d_lista = [ [ [1,2],[4,3],[7,4] ] , [ [2,11],[9,6],[7,5] ] , [ [1,15],[3,19],[0,2] ] , [ [9,20],[6,14],[9,8] ] ]

print(vector_lista[1])

print(array2d_lista[0][1])

print(array3d_lista[0][1][1])
```

```
Consola ×

>>> %Run -c $EDITOR_CONTENT

  2
  3.0
  3

>>>
```

Otros ejemplos:



Arrays. import

Utilizaremos el módulo estándar array

El módulo **array** define una estructura de datos de secuencia que se ve muy parecida a una **list**, excepto que todos los miembros tienen que ser del mismo tipo primitivo. Los tipos admitidos son todos numéricos u otros tipos primitivos de tamaño fijo como bytes.

```
import array as arr
num = arr.array('i', [1, 2, 3, 4, 5, 6])
```

En el ejemplo, se importa el módulo **array**, y se utiliza la instrucción **as** y luego **arr**, la lectura seria al módulo **array** lo vamos a nombrar como **arr**, luego creamos un **array** y fíjese se incorpora la letra **i**, este es el tipo de datos a utilizar existen otros tipos de datos.

Crear un array vacío, Modificar y agregar elementos

Primero se importa el módulo **array** y para facilitar el manejo lo nombramos como **arr** luego procedemos a instanciar una variable **array** del tipo entero con signo, para eso utilizamos 'i', y para el segundo parámetro introducimos una lista vacía **[]**

```
import array as arr
vector = arr.array('i',[])

print(vector)

array('i')
```

Las matrices son mutables; sus elementos se pueden cambiar de forma similar a las listas.

```
import array as arr

num = arr.array('i',range(0,11,1))
print(num)
num[0] = 100
print(num)
```

```
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

array('i', [100, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Para agregar un elemento a la matriz se utiliza el método **append()**,

```
import array as arr

num = arr.array('i',range(0,11,1))
print(num)
num.append(200)
print(num)
```

```
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 200])
```

y para agregar varios elementos usando el método **extend()**.

```
import array as arr

num = arr.array('i',range(0,11,1))
print(num)
num.extend([300,400,500,600])
print(num)
```

```
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 300, 400, 500, 600])
```

Arrays. Eliminar elementos

Podemos eliminar uno o más elementos de una matriz utilizando la instrucción ***del***

```
num = arr.array('i',range(25,35,1))
print(num)
del num[3]
print(num)
```

```
array('i', [25, 26, 27, 28, 29, 30, 31, 32, 33, 34])
array('i', [25, 26, 27, 29, 30, 31, 32, 33, 34])
```

Podemos usar el método **remove()** para eliminar el elemento dado

```
import array as arr

num = arr.array('i',range(25,35,1))
print(num)
num.remove(28)
print(num)
```

```
array('i', [25, 26, 27, 28, 29, 30, 31, 32, 33, 34])
array('i', [25, 26, 27, 29, 30, 31, 32, 33, 34])
```

y el método **pop()** para eliminar un elemento por medio del índice

```
import array as arr

num = arr.array('i',range(25,35,1))
print(num)
num.pop(6)
print(num)
```

```
array('i', [25, 26, 27, 28, 29, 30, 31, 32, 33, 34])
array('i', [25, 26, 27, 28, 29, 30, 32, 33, 34])
```

Funciones.

¿Qué es una función? ¿Cómo la ejecuto?

Una función es un grupo de declaraciones e instrucciones (código) relacionadas que realizan una tarea específica.

El objetivo de una función es no repetir trozos de código durante nuestro programa y reutilizar el código para distintas situaciones. Sintaxis de una función:

```
def nombre_Funcion( parametros ):
    ''' cadena de domumentacion '''
    instrucciones
    instrucciones
    instrucciones
    return
```

¿Cómo hago para ejecutar(llamar) a una función?

Para ejecutar, invocar o llamar a una función tendremos que escribir su nombre seguido de paréntesis. Debemos prestar atención si la función necesita valores (parámetros) o no. Las funciones pueden ser llamadas desde un programa u otra función.

```
def saludo():
    #Cuerpo de la función

    #Instrucciones
    print("Hola usuario, le damos la bienvenida...\n")


#Llamamos la función
saludo()
```

```
Hola usuario, le damos la bienvenida...
```

Una función en Python es una colección de declaraciones conectadas que realizan una sola tarea. Las funciones ayudan en la división de nuestro programa en porciones modulares más pequeñas. Las funciones ayudan a que nuestro programa se vuelva más ordenado y controlable a medida que crece en tamaño. También elimina la repetición y hace que el código sea reutilizable.

Nombre de la función: debe respetar la nomenclatura para instanciar variables, funciones, constantes y clases.

Funciones integradas que utilizamos hasta el momento:

- len
- range
- print
- if
- while

Funciones. Devolución de valores (Return).

La declaración **return** es una declaración especial que se utiliza dentro de una función o método para devolver el resultado de la función cuando es llamada.

Sintaxis: **return** *valor de retorno* que puede ser **int**, **str**, **float**, **list**, **tuple**, **dict**, **set**. entre otros

Puede omitir el valor de retorno de una función y usar **return** sin valor de retorno o también puede omitir **return**. En estos casos, el valor devuelto será **None**.

```
def suma():
    a = 45
    b = 56
    return a + b

valor = suma()
print(f"El resultado de la suma es {valor}")
```

```
El resultado de la suma es 101
```

```
def suma():
    return 45 + 56

print(f"El resultado de la suma es {suma()}")
```

```
El resultado de la suma es 101
```

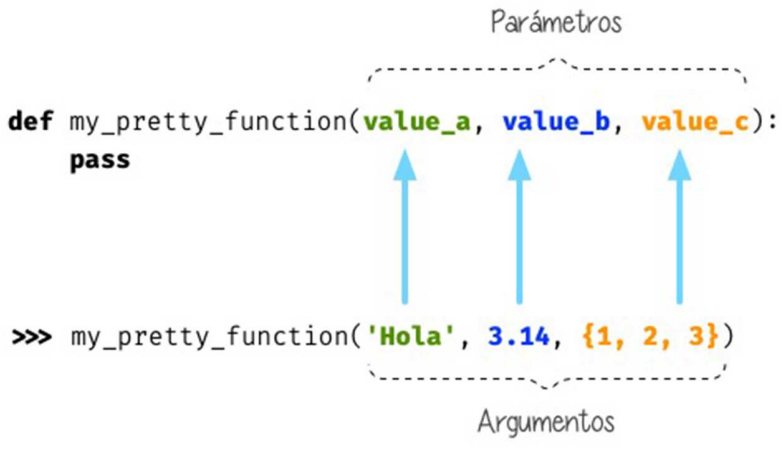
```
def suma():
    resultado = 45 + 56

valor = suma()
print(f"El resultado de la suma es {valor}")
```

```
El resultado de la suma es None
```

No confundir **return** con **print()**. El valor de retorno de una función nos permite usarlo fuera de su contexto. El hecho de añadir **print()** al cuerpo de una función es algo «coyuntural» y no modifica el resultado de la lógica interna.

Funciones. Parámetros y argumentos

Se pueden crear funciones que requieran o reciban uno o más valores, cuando se crea la función se utilizan los parámetros para luego utilizarlos en la función. Cuando se llama a la función se cargan los valores (argumentos) para cumplir con los parámetros de nuestra función.

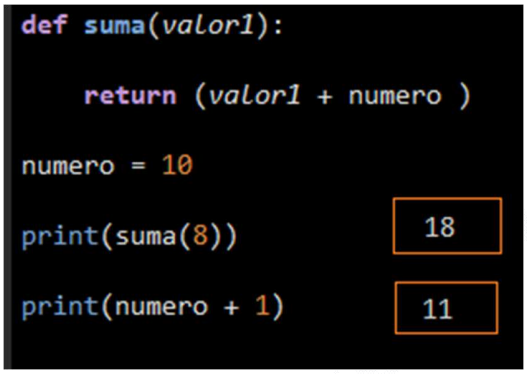


Delgado Quintero, DQS (2022). *Parámetros y argumentos de una función* , https://aprendepython.es

Funciones. Alcance de las variables

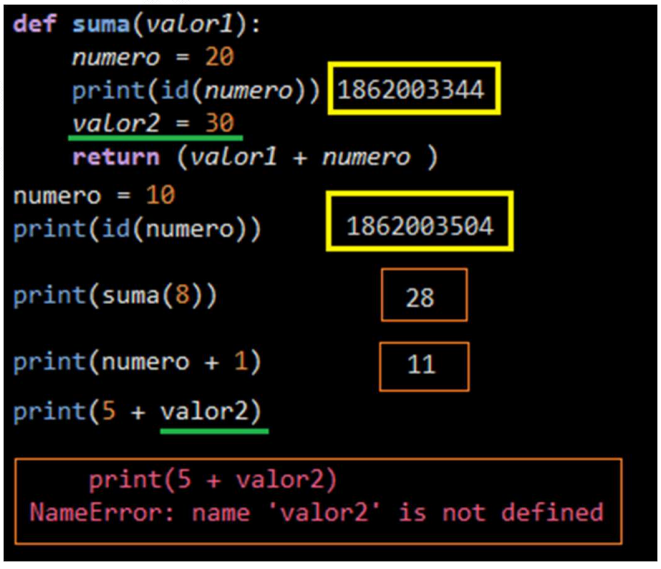
Variables globales

Cualquier variable declarada fuera de la función se denomina variable global. Tal variable tiene alcance global, es decir, es accesible desde cualquier lugar, ya sea dentro o fuera de la función.



Variables locales

Una variable declarada dentro del cuerpo de la función se denomina variable local. Se dice que una variable de este tipo tiene un alcance local, es decir, sólo es accesible dentro de la función y no fuera de ella.



Funciones.

Parámetros y argumentos - Ejercicios

Ejercicio 1: Escriba un programa que contenga una función con 1 parámetro, llamada **saludo_al_usuario**, la función deberá poder recibir una nombre (**string**), y luego deberá imprimir "Hola ..argumento(nombre)..buen día"

```
def saludo_al_usuario(nombre):
    print(f"Hola {nombre} buen día")

saludo_al_usuario('Alejandro')
```

```
Hola Alejandro buen día
```

Ejercicio 2: Escriba un programa que contenga una función con 2 parámetros, llamada multiplicación, la función deberá poder recibir dos números enteros (**int**), y luego debe realizar la multiplicación y retornar el valor, luego debe mostrarla.

```
def multiplicacion(num1,num2):
    return num1 * num2

resultado = multiplicacion(8,6)
print(f" 8 x 6 = {resultado} ")
```

```
8 x 6 = 48
```

```
def multiplicacion(num1,num2):
    return num1 * num2

print(f" 8 x 6 = {multiplicacion(8,6)} ")
```

```
8 x 6 = 48
```

Ejercicio 4: Escriba una función para cargar un vector de **n** cantidad de elementos con valor **0** cero, debe retornar el vector. Luego escriba un programa que llame a esa función y muestre el vector por pantalla.

```
def crea_vector(elementos):

    import array as arr
    vector = arr.array('i',[])

    for i in range(elementos):
        vector.append(0)
    return vector

print(crea_vector(5))

print(*crea_vector(5))
```

```
array('i', [0, 0, 0, 0, 0])

0 0 0 0 0
```

Funciones Recursivas

¿Qué es la recursividad?

La recursividad es un método para resolver problemas que implica descomponer un problema en subproblemas más y más pequeños para que pueda resolverse trivialmente. Por lo general, la recursividad implica una función que se llama a sí misma hasta llegar a un punto de salida. ¡Cada subproblema se va apilando, si! se genera una pila y luego se va resolviendo utilizando el método de pila FIFO el primero en entrar es el último en salir, por lo tanto, se van resolviendo los subproblemas desde el último hasta el primero.

Cualquier función recursiva tiene dos secciones de código claramente divididas:

Debe llamarse a sí mismo

Debe tener al menos una condición clara de fin o caso base del problema, puede tener más de una de ser necesario. Es muy importante porque de lo contrario, la función se llamaría de manera indefinida.

Este modelo recursivo se lo debe pensar como una función recursiva que servirá para dar solución a ciertos problemas, No cualquier problema puede resolverse de forma recursiva.

Es importante aclarar que dentro de una función o algoritmo recursivo no deben existir ciclos o bucles, salvo algunos casos muy particulares para resolver alguna operación transversal a la función.

¿Por qué utilizar la recursividad?

La secuencia de Fibonacci es una serie de números en la que cada número es la suma de los dos números anteriores. La secuencia comienza con 0 y 1, y los siguientes números son la suma de los dos números anteriores. Por lo tanto, la secuencia comienza así: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Este modelo recursivo se lo debe pensar como una función recursiva que servirá para dar solución a ciertos problemas –estos son casos particulares donde la naturaleza del problema es recursiva–.

No cualquier problema puede resolverse de forma recursiva, aunque muchas veces puede implementarse una solución recursiva para problemas que no son de esta naturaleza.

Ejercicio:

Implementación de una función que permita obtener el valor en la sucesión de Fibonacci para un número dado

Sin recursividad

```
def fibonacciI(n):
    n0 = 0
    n1 = 1
    if n==0:
        print("0")
        fib=n
    elif n==1:
        print("0 1")
        fib=n
    else:
        print("0 1",end= ' ')
        i=2
        while(i<=n):
            fib = n0 + n1
            print(fib,end=" ")
            n0=n1
            n1=fib
            i+=1
    return fib

a=fibonacciI(10)

print("\nValor:",a)
```

```
0 1 1 2 3 5 8 13 21 34 55
Valor:  55
```

Con recursividad

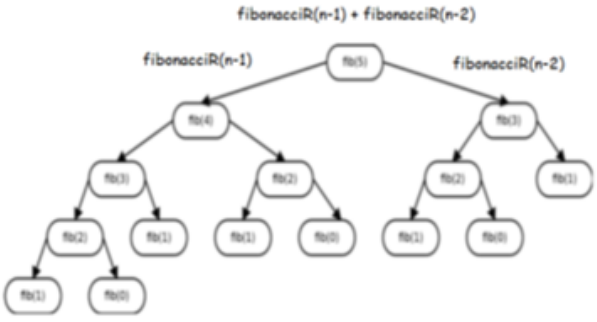
```
def fibonacciR(n):
    if(n<=1):
        return n
    else:
        return fibonacciR(n - 1) + fibonacciR(n - 2)

sucesiones = 10
for i in range(sucesiones+1):
    print(fibonacciR(i),end = " ")


print(f"\nValor: ", fibonacciR(sucesiones))
```

```
0 1 1 2 3 5 8 13 21 34 55
Valor:  55
```

Con recursividad

```
def fibonacciR(n):
    if(n<=1):
        return n
    else:
        return fibonacciR(n - 1) + fibonacciR(n - 2)

sucesiones = 10
for i in range(sucesiones+1):
    print(fibonacciR(i),end = " ")


print(f"\nValor: ", fibonacciR(sucesiones))
```

```
0 1 1 2 3 5 8 13 21 34 55
Valor:  55
```

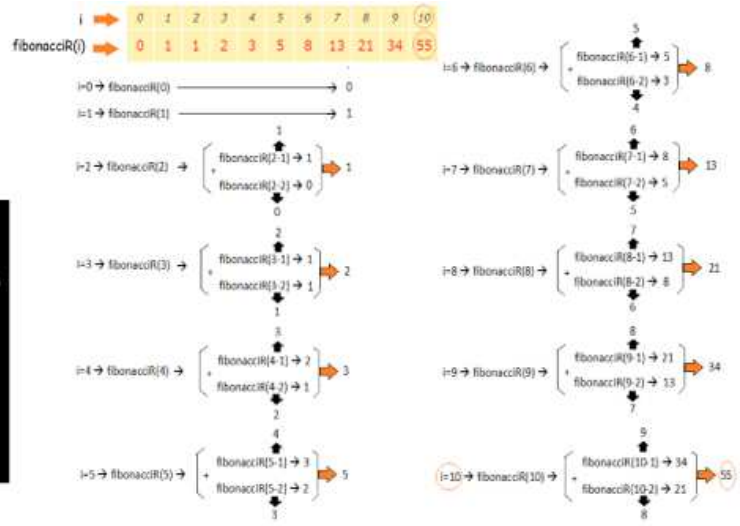


Implementación de una función que permita obtener la serie y el valor en la sucesión de Fibonacci para un número dado.

Con recursividad

```
def fibonacciR(n):
    if(n<=1):
        return n
    else:
        return fibonacciR(n - 1) + fibonacciR(n - 2)

sucesiones = 10
for i in range(sucesiones+1):
    print(fibonacciR(i),end = " ")

print(f"\nValor: ", fibonacciR(sucesiones))
```

```
0 1 1 2 3 5 8 13 21 34 55
Valor:  55
```



Resolución de un problema sin utilizar funciones recursivas

Calcular la suma de una lista de números como, por ejemplo: [1,3,5,7,9] Utilice una función que se llame sumalista(parámetro) y acepte la lista como argumento, retorna el valor de la suma.

```
def sumalista(listaNumeros):
    laSuma = 0

    for i in listaNumeros:
        # Mostramos el procedimiento
        print(f"{laSuma} + {i} = {laSuma + i}")

        laSuma = laSuma + i

    return laSuma

print(sumalista([1,3,5,7,9]))
```

```
0 + 1 = 1
1 + 3 = 4
4 + 5 = 9
9 + 7 = 16
16 + 9 = 25


25
```

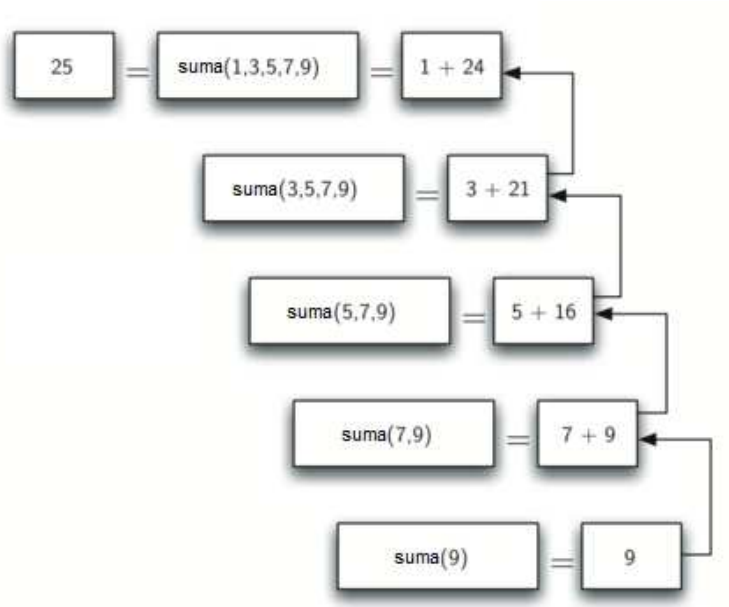
Resolución de un problema sin utilizar funciones recursivas

Solución del ejercicio 1. Calcular la suma de una lista de números como, por ejemplo: [1,3,5,7,9] Utilice una función que se llame sumalista(parámetro) y acepte la lista como argumento retorne el valor de la suma. Opcional imprima cada suma, por ejemplo:

Imaginemos por un minuto que usted no tiene ciclos while o for. ¿Cómo calcularía la suma de una lista de números? Si usted fuera un matemático,

Empezaría a sumar parejas de números separando en términos por medio de paréntesis, podríamos reescribir la lista como una expresión completamente agrupada. Tal expresión tiene el siguiente aspecto: (1+(3+(5+(7+9))))

La siguiente Figura muestra la serie de llamadas recursivas que se necesitan para sumar la lista [1,3,5,7,9] y las sumas que se realizan a medida que sumalista funciona hacia atrás a través de la serie de llamadas. Cuando sumalista devuelve el resultado del problema superior, tenemos la solución de todo el problema.



```
def sumalista(listaNumeros):
    if len(listaNumeros) == 1:

        print(listaNumeros[0])

        return listaNumeros[0]
    else:
        print(f"{listaNumeros[0]} + {listaNumeros[1:]}")

        return listaNumeros[0] + sumalista(listaNumeros[1:])

print(sumalista([1,3,5,7,9]))
```

```
1 + [3, 5, 7, 9]
3 + [5, 7, 9]
5 + [7, 9]
7 + [9]
9


25
```

Primero debemos definir la función que vamos a utilizar y sus parámetros, en la línea 2 estamos comprobando si la lista es de longitud un elemento. Esta comprobación es crucial y es nuestra cláusula de escape de la función return. El else , entrara si hay más de un elemento donde se genera la suma en la línea 9 nuestra función se llama a sí misma! Ésta es la razón por la que llamamos recursivo al algoritmo sumalista. Una función recursiva es una función que se llama a sí misma.

Pensemos en esta serie de llamadas como una serie de simplificaciones. Cada vez que hacemos una llamada recursiva estamos resolviendo un problema más pequeño, hasta llegar al punto en el que el problema no puede ser más pequeño. Cuando llegamos a este punto, comenzamos a juntar las soluciones de cada uno de los pequeños problemas hasta que el problema inicial se resuelva.

Las tres leyes de la recursividad

- Un algoritmo recursivo debe tener un caso base.
- Un algoritmo recursivo debe cambiar su estado y moverse hacia el caso base.
- Un algoritmo recursivo debe llamarse a sí mismo, recursivamente

```
def sumalista(listaNumeros):

    # Primer ley
    if len(listaNumeros) == 1:  # Caso base

        return listaNumeros[0]

    else:
        # Segunda Ley: listaNumeros[1:] Estamos achicando la lista en cada llamada
        # Tercera ley: llamamos a la propia función
        print(listaNumeros[1:])
        return listaNumeros[0] + sumalista(listaNumeros[1:])

print(sumalista([1,3,5,7,9]))
```

Veamos si se aplicaron las 3 leyes en la función recursiva sumalista.

En primer lugar, un caso base es la condición que permite que el algoritmo detenga la recursividad. dijimos que (Un caso base es típicamente un problema que es lo suficientemente pequeño como para resolverlo directamente). En el algoritmo sumalista, el caso base es una lista de longitud 1 y se controla con una comparación dentro de un condicional (if).

La segunda ley, debemos organizar un cambio de estado que mueva el algoritmo hacia el caso base. Un cambio de estado significa que se modifican algunos datos que el algoritmo está usando. Por lo general, los datos que representan nuestro problema se hacen más pequeños de alguna manera a resolver.

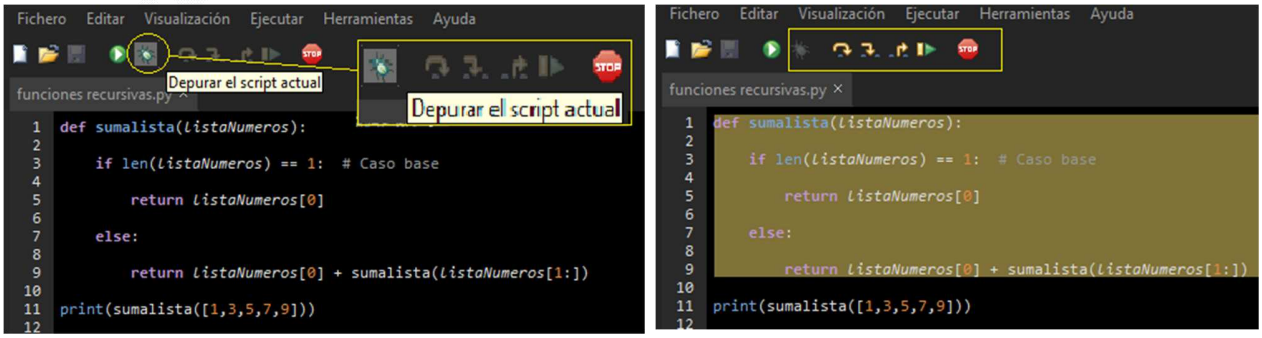
Por ejemplo en el algoritmo sumalista nuestra estructura de datos primaria es una lista, así que debemos centrar nuestros esfuerzos de cambio de estado en la lista. Dado que el caso base es una lista de longitud 1, una progresión natural hacia el caso base es acortar la lista. Esto es exactamente lo que ocurre en la línea 9 del cuando llamamos a sumalista con una lista más corta sumalista(listaNumeros[1:]. Con [1::] se va a acortando la lista con cada recursión.

La última ley es que el algoritmo debe llamarse a sí mismo. Esta es la definición misma de la recursividad. La recursividad es un concepto confuso para muchos programadores principiantes. Cuando hablamos de recursividad, puede parecer que estamos hablando en círculos. Tenemos un problema que resolver con una función, ¡pero esa función resuelve el problema llamándose a sí misma! Pero la lógica no es circular en absoluto; la lógica de la recursividad es una expresión elegante de resolver un problema al descomponerlo en problemas más pequeños y más fáciles.

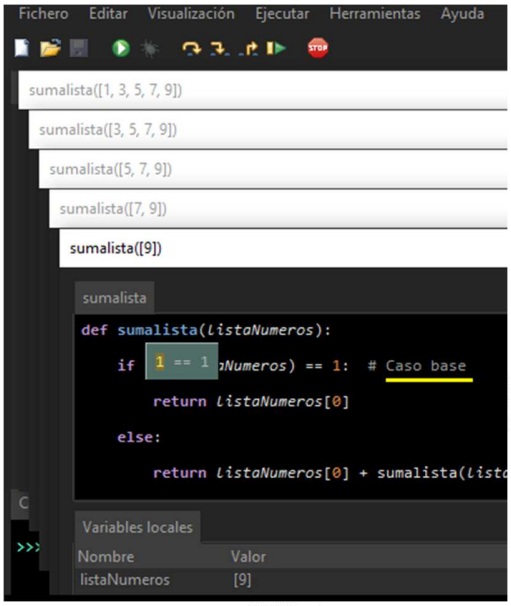
Depurando la función

La depuración es el proceso de encontrar y solucionar errores en el código fuente de cualquier software y también se utiliza para monitorear los procesos. Los programadores de computadoras estudian el código esta manera más a detalle.

Hacemos en el icono que parece un bicho , desde aquí empezamos a depurar el programa, cuando presionen el botón se iluminará la parte el código a ejecutarse.

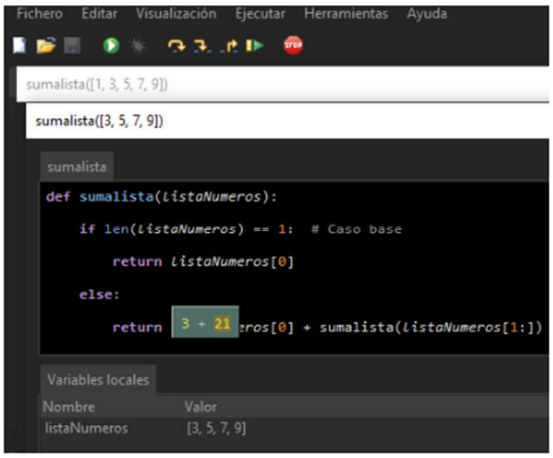


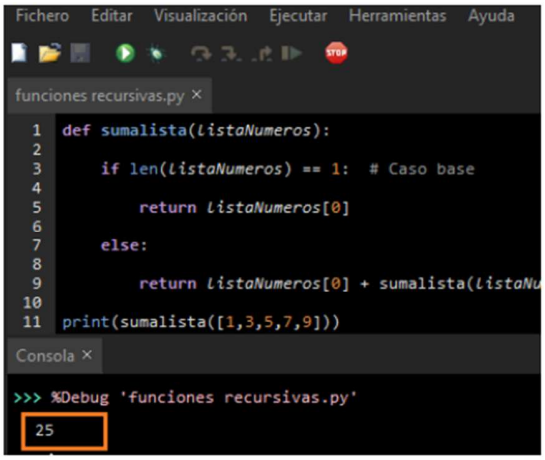
Para continuar tenemos distintos botones luego en casa pueden ir analizando cada uno de ellos para su conocimiento. por el momento nosotros solo presionaremos la tecla F7 de nuestro teclado, y empezará a ejecutarse el código que se va iluminando siguiendo los pasos hasta resolver cada línea de código.



Primero ejecuta la línea que llama a la función la línea 13 y presionamos F7 varias veces, en este paso python está reconociendo la lista elemento por elemento. al terminar de recorrer toda la lista, se abrirá otra ventana con nuestro código pero esta vez se iluminará el código de la función y empezará a recorrer la misma.

Seguimos presionando F7 observen cada línea y código que se ilumina, vemos que se van desarrollando y resolviendo paso a paso el código. Al llegar a la problema más pequeño que es el **caso base** imagen 3, la recursión comienza a resolver todos las sumas, en la imagen 4 vemos que se van cerrando las ventanas que se fueron abriendo hasta llegar el resultado final imagen 5.





Cuenta regresiva hasta cero a partir de un número

```
def cuenta_atras(num):
    num -= 1
    if num > 0:
        print(num)
        cuenta_atras(num)
    else:
        print("Booooooooom!")
    print("Fin de la función", num)

cuenta_atras(5)
```

```
4
3
2
1
Booooooooom!
Fin de la función 0
Fin de la función 1
Fin de la función 2
Fin de la función 3
Fin de la función 4
```

Algoritmos y Estructuras de Datos III

Tecnicatura Superior en Análisis de Sistemas