# FICHA 1 : FUNDAMENTOS DE PROGRAMACIÓN

Computadoras un gran invento, que no para de mostrar innovaciones.
Las primeras aparecieron en la década del 40, en el siglo XX(20).

Primera persona que propuso el concepto de máquina de cálculo (fue una calculadora) charles babbage en 1821 y junto a ada byron crearon la máquina de babbage y byron mas conocida como ‘“Analytical Engine” o “motor analítico” .
En 1890 el ing. El estadounidense Herman Hollerith diseñó una máquina para procesar los datos del censo en forma más veloz, la cual tomaba los datos desde una serie de tarjetas perforadas (que ya habían sido sugeridas por Babbage para su Analytical Engine). Puede decirse que la máquina de Hollerith fue el primer antecedente que efectivamente llegó a construirse de las computadoras modernas, Herman es el creador de IBM( International Business Machines) .
En 1943 luego de la 2da guerra mundial alan turing creó una maquina desencriptadora que se llamó Bombe.
Turing recurrió a Tommy Flowers para que diseñe y construya una nueva máquina, a la cual llamaron Colossus.
La primera máquina que podemos llamar computadora en el sentido moderno, apareció en 1944 de la mano del ingeniero estadounidense Howard Aiken

# ALGORITMOS Y PROGRAMAS

Definición de computadora : se trata de un aparato capaz de ejecutar una serie de órdenes que permiten resolver un problema. La serie de órdenes se designa como programa, y la característica principal del computador es que el programa a ejecutar puede ser cambiado para resolver problemas distintos

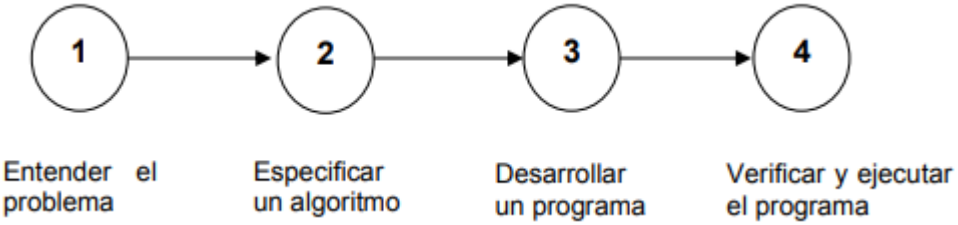
Algoritmo : el conjunto finito de pasos para resolver un problema
Si un algoritmo se plantea y escribe de forma que pueda ser cargado en una computadora, entonces tenemos un programa


En última instancia una computadora es lo que se conoce como una “máquina algorítmica”, pues su capacidad esencial es la de poder seguir paso a paso un algoritmo dado.
Los algoritmos son una parte fundamental del proceso de programación.

Los algoritmos se caracterizan principalmente porque son:

- Precisos: La palabra preciso indica que se distingue con claridad, de igual manera un algoritmo es claro en cada uno de sus pasos.
- Definido: El algoritmo está delimitado, sólo procesa la información y las operaciones que tiene, no realiza operaciones “fantasma”, si se realiza el algoritmo dos o más veces con los mismo datos siempre dará el mismo resultado.
- Finito: Refiriéndose esta característica a que un algoritmo siempre va a tener un fin, .
- mediante esta característica el algoritmo se culmina la estructura de un algoritmo.

Pass para resolver un problema mediante un computador



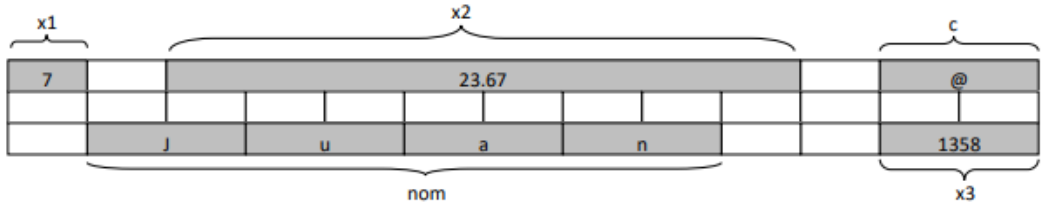

## FUNDAMENTOS DE REPRESENTACIÓN DE INFORM. EN SISTEMA BINARIO

Una computadora sólo sirve si posee un programa cargado al que pueda interpretar y ejecutar instrucción por instrucción.
Esto implica que una computadora de alguna forma es capaz de representar y retener información dentro de sí. En ese sentido, se llama memoria al dispositivo interno del computador en el cual se almacena esa información y básicamente, la memoria puede considerarse como una gran tabla compuesta por celdas individuales llamadas bytes, de modo que cada byte puede representar información usando el sistema de numeración binario basado.

## ELEMENTOS BÁSICOS DE PROGRAMACIÓN EN PYTHON : TIPOS,VARIABLES Y ASIGNACIONES:

La cantidad de memoria que ocupan  los valores(datos y resultados ) dependerá de que tipo de tipo de datos esté usando ejemplo, valor entero, valor de coma flotante,  o un valor booleano (true o false)
Una variable es un grupo de bytes asociado a un nombre o identificador, de tal forma que a través de dicho nombre se puede usar o modificar el contenido de los bytes asociados a la variable. (x1  = 7)

**Figura 7: Memoria de un computador (esquema)**




**: Tabla de tipos de datos elementales en Python (sólo los más comunes).**

| Tipo (o Clase) | Descripción | Bytes por cada variable | Rango |
|---|---|---|---|
| **bool** | valores lógicos | 1 | [False, True] |
| **int** | números enteros | dinámico[7] | ilimitado[8] |
| **float** | números reales | 8 | hasta 15 decimales |
| **str** | cadenas de caracteres | 2 * cantidad de caracteres | Unicode |

• El nombre o identificador de una variable en Python, sólo puede contener letras del alfabeto inglés (mayúsculas y/o minúsculas, o también dígitos (0 al 9), o también el guión bajo ( _ ) (también llamado guión de subrayado).

• El nombre de una variable no debe comenzar con un dígito.

• Las palabras reservadas del lenguaje Python no pueden usarse como nombres de variables.

• El nombre de una variable puede contener cualquier cantidad de caracteres de longitud.

• Recordar que Python es case sensitive: Python hace diferencia entre minúsculas y mayúsculas, por lo que toma como diferentes a dos nombres de variables que no sean exactamente iguales. El identificador sueldo no es igual al identificador Sueldo y Python tomará a ambos

## OPERADORES ARITMÉTICOS

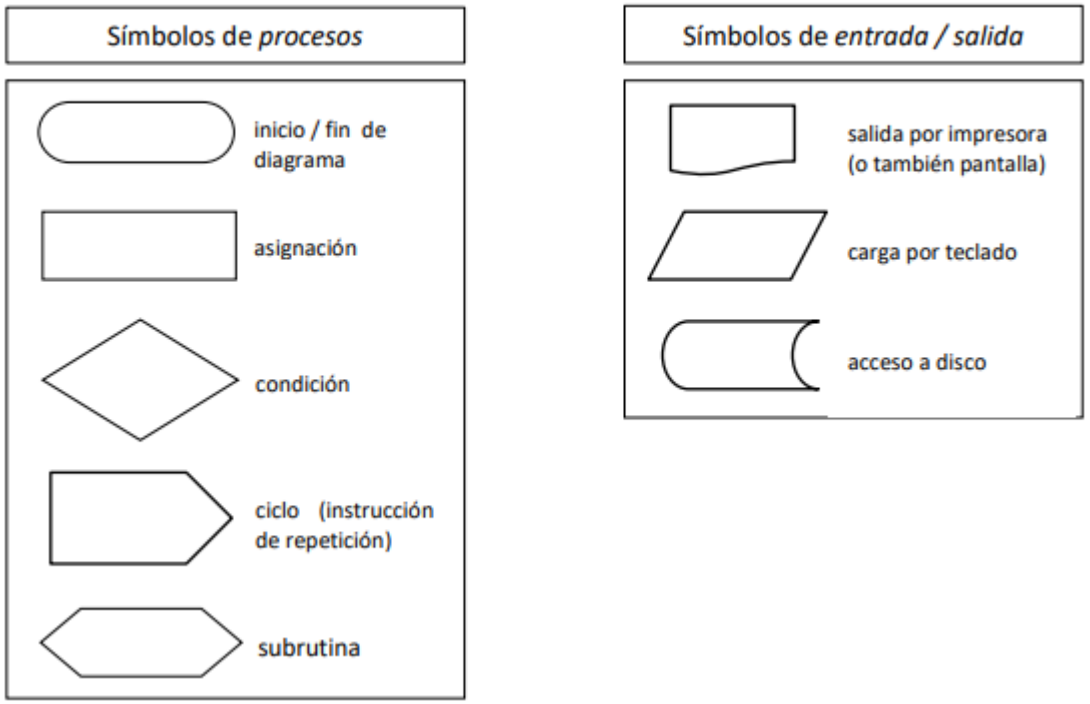
| Operador | Significado | Ejemplo de uso |
|---|---|---|
| + | suma | a = b + c |
| - | resta | a = b – c |
| * | producto | a = b * c |
| / | división de coma flotante | a = b / c |
| // | división entera | a = b // c |
| % | resto de una división | a = b % c |
| ** | potencia | a = b ** c |

## FICHA 2 : ESTRUCTURAS SECUENCIALES

DIAGRAMAS DE FLUJO
Es un gráfico que permite representar en forma clara y directa el algoritmo para resolver un problema. Se basa en el uso de unos pocos símbolos unidos por líneas rectas descendentes

Los diagramas de flujo y el pseudocódigo son dos de estas técnicas que ayudan a hacer unívoca la representación del algoritmo, ellas son técnicas auxiliares que ayudan al programador a entender un poco más lo que quiere plasmar en su código.

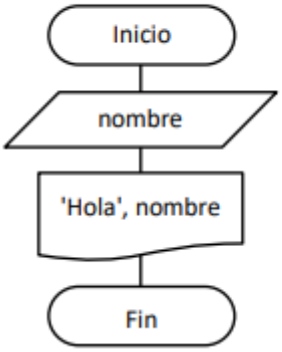
**Figura 3: Principales símbolos usados en un diagrama de flujo.**




El pseudocódigo se trata de una herramienta por medio de la cual se plantea el algoritmo sin usar gráficas, escribiendo cada acción o paso en lenguaje natural o cotidiano

ejemplo de un problema de la ficha:

**Figura 5: Diagrama y pseudocódigo para el problema de cargar el nombre y mostrar un saludo**



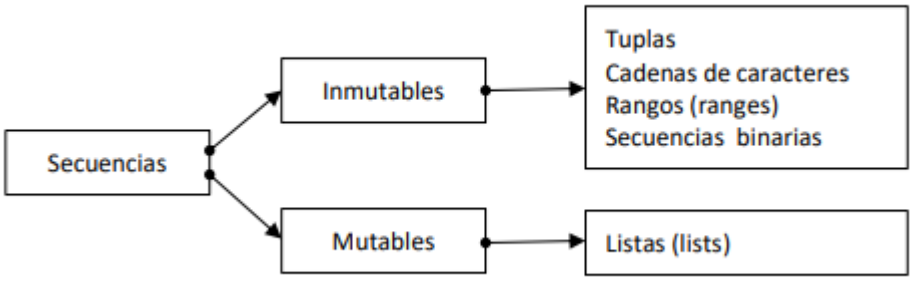

```
Algoritmo:
    1. Cargar nombre: nombre del usuario
    2. Mostrar 'Hola' y nombre
```

# FICHA 3 : TIPOS DE ESTRUCTURADOS BÁSICOS

## TIPOS DE SECUENCIA DE DATOS:

Tipos de datos simples ---> float,int,bool.
Tipos de datos compuestos (pueden contener varios valores al mismo tiempo)--->cadena de caracteres, registros,etc.
después de esos vienen las secuencias que pueden ser:

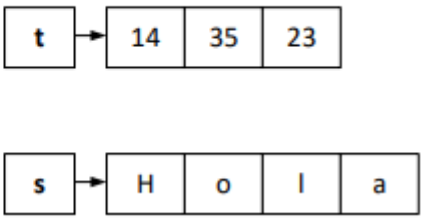
**Tipos de *secuencias* en Python 3.**




Tupla : secuencia inmutable que puede contener distintos datos.va con parentesis para una tup vacia aveces no es necesario los paréntesis, separo los valores dentro de los paréntesis con coma, hay una función determinada en python que se llama tuple().

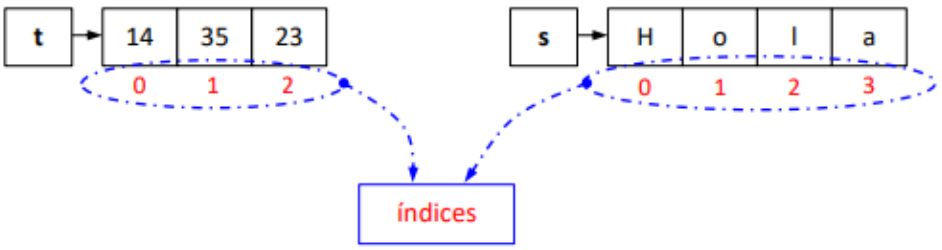
**Ejemplos gráficos de secuencias inmutables de tipo *cadena* y *tupla*:**

```
# una tupla…
t = 14, 35, 23
print(t)

# una cadena de caracteres…
s = 'Hola'
print(s)
```




pueden ir o no ir los paréntesis en t.
Función LEN() me dice la cantidad de elementos esa secuencia.

**Figura 3: Dos secuencias y sus casilleros identificados por índices.**




Para acceder a un elemento individual de una secuencia, basta con escribir *el identificador de la misma y luego agregar entre corchetes el índice del casillero a acceder*:

```
t = 14, 35, 23
print(t[1])      # muestra: 35
x = t[2]
print(x)         # muestra: 23

s = 'Hola'
print(s[0])      # muestra: H
c = s[3]
print(c)         # muestra: a
```

Mutable ----> se puede modificar un valor de un elemento de una secuencia.
Inmutable ----> no se puede modificar nada.

LA GUIA PEP

Se centra específicamente en recomendaciones de estilo de escritura de código fuente en Python.
Para indentar use 4(cuatro) espacios en lugar de tabs (tabuladores). El uso de tabuladores introduce confusión, y con 4 espacios de indentación tendrá el equilibrio justo entre "poco" y "demasiado".
- Corte sus líneas de código en no más de 79 caracteres por línea. Esto le ayudará a mantener visible gran parte del código fuente en pantallas pequeñas, y le permitirá mostrar varios archivos de código fuente uno al lado del otro en pantallas grandes.
- Use líneas en blanco para separar funciones, clases y largos bloques de código dentro de una función o una secuencia (veremos la forma de desarrollar funciones más adelante en el curso).
- Cuando sea posible, coloque sus comentarios en líneas específicas para esos comentarios (de ser posible, no los agregue en la misma línea de una instrucción). - Utilice cadenas de documentación (docstrings) [2]: mantenga actualizada la documentación de sus programas (veremos esto más adelante el curso).
- Coloque espacios alrededor de los operadores en un expresión, y después de las comas en una enumeración, pero no inmediatamente a los lados de cada paréntesis cuando los use: a = f(1, 2) + g(3, 4).
- No utilice codificaciones de caracteres demasiado extrañas si está pensando en que su código fuente se use en contextos internacionales. En cualquier caso, el default de Python (que es UTF-8) o incluso el llamado texto plano (ASCII) funciona mejor.

- Del mismo modo, evite el uso de caracteres no-ASCII en el nombre de un identificador (por ejemplo, en el nombre de una variable) si existe aunque sea una pequeña posibilidad de que su código fuente sea leído y mantenido por personas que hablen en otro idioma.
- En cuanto al uso de comillas dobles o simples para delimitar cadenas de caracteres, no hay una recomendación especial. Simplemente, mantenga la consistencia: si comenzó con una forma de hacerlo, apeguese a ella y no la cambie a cada momento. Cuando necesite incluir un tipo de comilla (simple o doble) en una cadena, use la otra para delimitarla (en lugar de caracteres de control que restan legibilidad).
- Si tiene operadores con diferentes prioridades, considere agregar espacios alrededor de los que tengan menor prioridad, y eliminar los espacios en los que tengan prioridad mayor. Use el sentido común. Sin embargo, nunca coloque más de un espacio y siempre coloque la misma cantidad de espacios antes y después del mismo operador.

Si:
```
m = p + 1
t = x*2 - 1
h = x*x + y*y
c = (a+b) * (a-b)
```

No:
```
p=p+1
t= x*2-1
h = x * x + y * y
c = (a + b) * (a - b)
```

- Como ya se indicó, en Python no hay una recomendación especial en cuanto a convenciones de nombres o identificadores. En general, use sentido común: sea cual sea la convención que siga el programador, debe ser claramente distinguible y mantenerse en forma coherente. Otra vez, digamos que en el desarrollo de las Fichas de Estudio de este curso y en los ejemplos de ejercicios y programas resueltos que se entreguen a los estudiantes, aplicaremos las siguientes convenciones en cuanto a nombres de variables (y de nuevo, aclaramos que los estudiantes e incluso sus profesores podrán usar otras convenciones reconocidas si las prefieren. Sólo se sugiere que se mantengan consistentes con ellas):

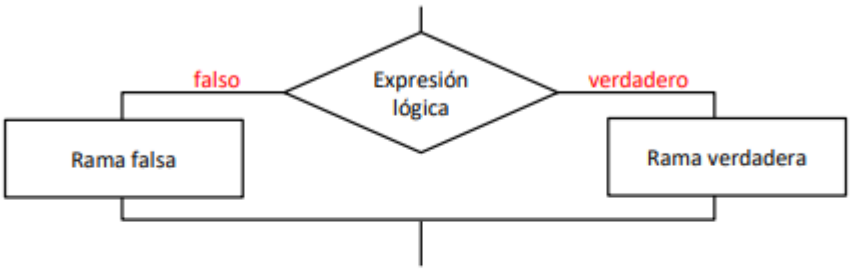
▪ El nombre de una variable se escribirá siempre en minúsculas (Ejemplos: sueldo, nombre, lado).
▪ Cuando se quiera que el nombre de una variable agrupe dos o más palabras, usaremos el guión bajo como separador (Ejemplos: mayor_edad, horas extra).
▪ Excepcionalmente designaremos en mayúsculas sostenidas a alguna variable que represente algún valor muy conocido y normalmente escrito en mayúsculas (Ejemplos: PI (por el número pi), o IVA (por nuestro bendito impuesto al valor agregado)).

- Nunca utilice la l ('letra ele' minúscula) ni la O ('letra o' mayúscula) ni la I ('letra i' mayúscula) como nombre simple de una variable: en algunas fuentes de letras esos caracteres son indistinguibles de los números 1 y 0 y obviamente causaría confusión.

## FICHA 4: ESTRUCTURAS CONDICIONALES

Una instrucción condicional contiene una expresión lógica que puede ser evaluada por verdadera o por falsa, y dos bloques de instrucciones adicionales designados en general como la salida o rama verdadera y la salida o rama falsa.

**Figura 1: Diagrama general de una instrucción condicional típica.**




En el lenguaje Python, una instrucción condicional típica como la que mostramos en la figura anterior, se escribe (esquemáticamente) así [1]:

```
if expresión lógica:
      instrucciones de la rama verdadera
else:
      instrucciones de la rama falsa
```

FICHA 5 : ESTRUCTURAS CONDICIONALES : VARIANTES

Sin embargo, puede ocurrir (y de hecho es muy común) que para una condición sólo se especifique la realización de una acción si la respuesta es verdadera y no se requiera hacer nada en caso de responder por falso.
Una instrucción condicional de ese tipo se suele designar como condición simple, y en ella no se especifica la rama else: la instrucción condicional termina cuando termina la rama verdadera

```
if expresión lógica:
      instrucciones de la rama verdadera

continuación del programa
```

Hay muchos casos de estudio en esta ficha y en la 4 también.

EXPRESIONES DE CONTEO Y ACUMULACION.

Un contador es una variable que sirve para contar ciertos eventos que ocurren durante la ejecución de un programa. Intuitivamente, se trata de una variable a la cual se le suma el valor 1 (uno) cada vez que se ejecuta la expresión.

**Figura 4: Uso básico de un contador.**

```
__author__ = 'Catedra de AED'

a = 0
num = int(input('Ingrese un número: '))
if num < 0:
    a = a + 1
num = int(input('Ingrese otro: '))
if num < 0:
    a = a + 1
num = int(input('Ingrese otro: '))
if num < 0:
    a = a + 1
print('Cantidad de negativos cargados:', a
```

```
c = c + 2
d = d * 4
e = e / 3
```

# En estos contadores lo primero que se ejecuta es la parte derecha y con el valor actual del contador a, sí primero valía 0 luego vale a +1.

#En segundo lugar se asigna el valor así obtenido en la misma variable a, con lo cual se cambia el valor original. En este caso, la primera vez que se cargue un negativo en número la variable a quedará valiendo 1. Si se carga un segundo negativo en num, a quedará valiendo 2, y así cada vez que se ingrese un negativo, asignando a la variable a su valor anterior sumado en uno.

esto es de acuerdo el ejemplo anterior.

un acumulador es básicamente una variable que permite sumar los valores que va asumiendo otra variable o bien otra expresión en un proceso cualquiera.

es una variable que actualiza su valor en términos de su propio valor anterior y el valor de otra variable u otra expresión.(esta es otro def ver bien cual estudio y dejo en el resu)

**Uso básico de un acumulador.**

```
__author__ = 'Catedra de AED'

s = 0
x = int(input('Ingrese un número: '))
s = s + x
x = int(input('Ingrese otro: '))
s = s + x
x = int(input('Ingrese otro: '))
s = s + x

print('La suma de los valores cargados es:', s)
```

s = s + x == s += x (no olvidar eso )

VARIABLES CENTINELAS(BANDERAS)

Una bandera es una variable (típicamente booleana, aunque podría ser de cualquier otro tipo) cuyo valor se controla en forma metódica a lo largo de la ejecución de un programa, de forma que cada valor posible se asocia a la ocurrencia o no de un evento (por ejemplo, si el evento ocurrió la bandera se asigna en True, y mientras el evento no ocurra el flag se mantiene en False)

FICHA 6 : ESTRUCTURAS REPETITIVAS : EL CICLO WHILE

Una estructura repetitiva es una instruccion compuesta que permite la repeticion controlada de la ejecucion de cierto conjunto de instrucciones en un programa.
esa repeticion debe detenerse o continuar de acuerdo al valor de cierta condicion que se incluye en el ciclo.

Las estructuras repetitivas(while,for,etc) constan de dos partes :

*Bloque de acciones o cuerpo del ciclo que es el conjunto de instrucciones cuya ejecución se debe repetir.

*Cabecera del ciclo, que incluye una condición de control y/o elementos adicionales en base a los que se determina si el ciclo continúa o se detiene.

**Diagrama de flujo de un ciclo while (diagramación clásica)**

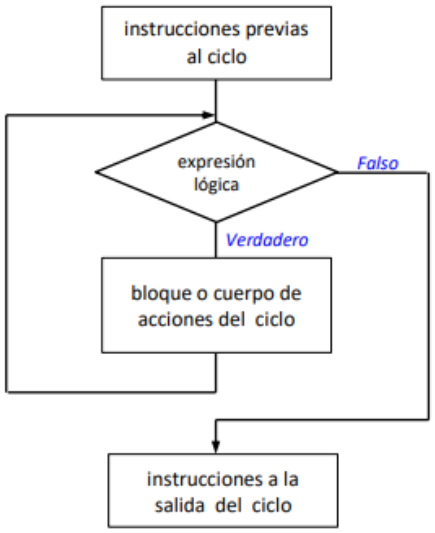
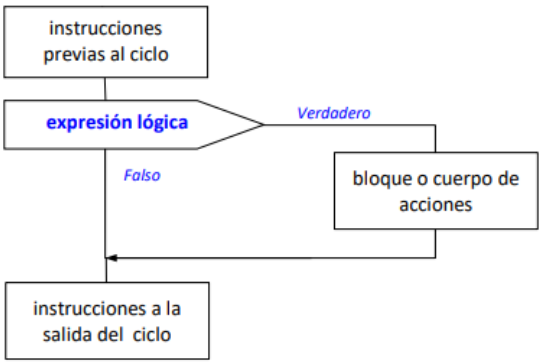



**Diagrama de flujo de un ciclo while (diagramación alternativa o indentada**



**En Python:**

```
while expresión_lógica:
        bloque de acciones

Siguiente instrucción fuera del ciclo
```

$$x_{1-2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

FICHA 7 : ESTRUCTURAS REPETITIVAS : EL CICLO FOR

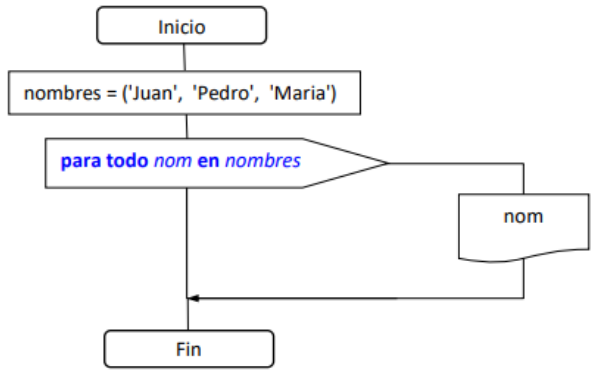
El CICLO FOR EN PYTHON

Este ciclo esta especialmente diseñado ´para recorrer secuencias como tuplas,cadena de caracteres,rangos,listas,etc.Recorriendo de a un elemento a la vez.

Este lo usamos casi siempre cuando sabemos previamente la cantidad de repeticiones a realizar.

Este esta espécialmente previsto para iterar estructura de datos.

**Figura 1: Diagrama de flujo del script para mostrar tres nombres con un ciclo for.**




nombre = (‘juan’,’pedro’,’maria’)

for nom in nombres():
print(nombre)

Ejemplo :
ciudad = ‘cordoba’
for c in ciudad:
print(c) #c va a ser mi variable iteradora, c va a tomar cada letra de la palabra

Si queremos realiza un rango(o range) de secuencia numerica designada. Una secuencia tipo range es una sucesion inmutable, ejemplo: sucesion numerica del 1 al 5 inclusive:
for i in range(1,6): [range(a,b) incluye el valor de a pero NO el de b]
print(i)
si llamo al rango con 1 solo parametro range(6) python asume que ese es el valor final SIN INCLUIRLO.

La funcion range acepta un tercer parámetro que indica el incremento a usar para pasar de un valor a otro, el incremento puede ser negativo o positivo.por defecto python asume q es 1

En general, un range *r* creado con la instrucción *r = range(start, stop, step)* representa una secuencia numérica inmutable que contiene sólo a los números dados por las siguientes dos expresiones [1]:

1. Si *step > 0* entonces el contenido de cada elemento *r[i]* se determina como:

```
r[i] = start + step*i   donde   i >= 0  y  r[i] < stop.
```

2. Si *step < 0* entonces el contenido de cada elemento *r[i]* se determina como:

```
r[i] = start + step*i   donde   i >= 0  y  r[i] > stop.
```

Por lo tanto, si *r = range(1, 10, 2)* entonces *r* representará una secuencia numérica de la forma (1, 3, 5, 7, 9) ya que según las fórmulas anteriores:

```
step = 2 es mayor a 0 por lo tanto:
r[0] = start + step*0 = 1 + 2*0 = 1   (y como 1 < stop (=10), se acepta)
r[1] = start + step*1 = 1 + 2*1 = 3   (y como 3 < stop (=10), se acepta)
r[2] = start + step*2 = 1 + 2*2 = 5   (y como 5 < stop (=10), se acepta)
r[3] = start + step*3 = 1 + 2*3 = 7   (y como 7 < stop (=10), se acepta)
r[4] = start + step*4 = 1 + 2*4 = 9   (y como 9 < stop (=10), se acepta)
```

En el siguiente ejemplo, entonces, el *ciclo for* muestra los primeros *n* números pares, comenzando desde el cero:

```
n = int(input('Cuantos pares quiere mostrar?: '))
for par in range(0, 2*n, 2):
    print(par)
```

en esta ficha entra tratamiento de caracteres pero es todo practico nomas.

FICHA 8 : ESTRCTURAS REPETITIVAS : VARIANTES

Ciclos en Python: Variantes y elementos de control adicionales

*Break : corta el ciclo de inmediato sin retornar a la cabecera.

```
suma, i = 0, 1
while i <= 5:
    n = int(input('Ingrese un número mayor a cero: '))
    if n <= 0:
          break
    suma += n
    i += 1

print('La suma de los números ingresados es:', suma)
```

En este ejemplo se cargan 5 numero, pero si llega a ingresar 0 a un num negativo al instante ese break corta el ciclo y da la suma de los num ingresados.

*Continue : Sirve para forzar una repeticion del ciclo sin terminar de ejecutar las instrucciones que queda por debajo de la invocacion continue.
para entender mejor uso el mismo ejercicio anterior pero con un continue , que hace que el ciclo no corte en este caso, aunque cargue negativos o ceros.

*Pass : Esta instrucción sirve para indicar al intérprete que simplemente considere vacío el bloque que la contiene.

```
# un ciclo que hace 10000 repeticiones sin bloque de acciones...
for i in range(10000):
    pass
```

Medición del tiempo de ejecución de un proceso en Python

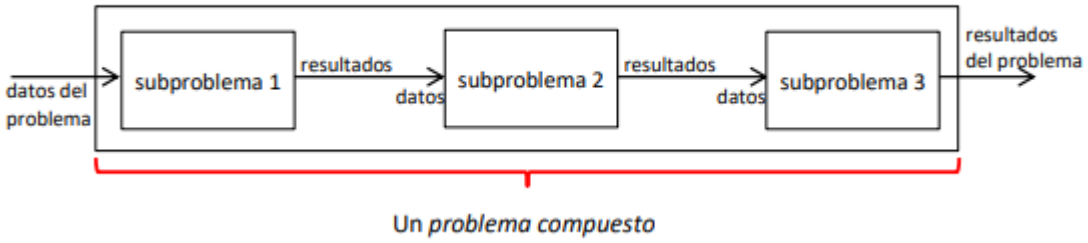
funcion : import time

## FICHA 9 : SUBPROBLEMAS Y FUNCIONES

Introducción al concepto de subproblema.

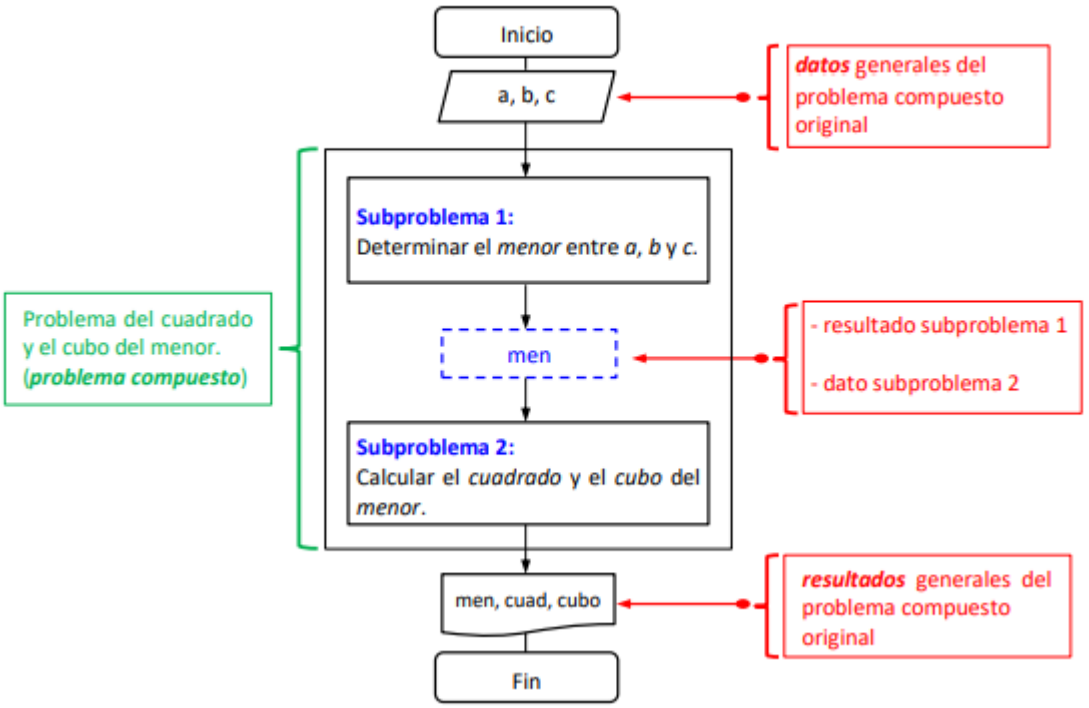
Un principio básico en el planteo de algoritmos para resolver problemas consiste justamente en trata de identificar los problemas simples de un problema compuesto considerando que cada su problema simple es también un problema cada emite los datos el desarrollo de sus procesos y genera resultados.

**Figura 1: Esquema general de un *problema compuesto* que incluye tres *subproblemas*.**




EJEMPLO DE SUBPROBLEMA LLEVADO A CABO DE UN EJERICICIO

**Figura 2: Modelo de subproblemas para el *problema del cuadrado y el cubo del menor*.**




Subrutinas: introducción y conceptos generales.

un subproblema un problema principal puede descomponerse en partes o subproblemas más sencillos para facilitar su planteo y luego unir las piezas para lograr el planteo final.

una subrutina (o subproceso) es un segmento de un programa que se escribe en forma separada del programa principal. A cada subrutina el programador asocia un nombre o identificador y mediante ese identificador la subrutina puede ser activada todas las veces que sea necesario.

Funciones en Python - Parámetros y retorno de valores

Funcion : un segmento de programa que se codifica en forma

separada, con un nombre asociado para poder activarla.

Una funcion tiene:

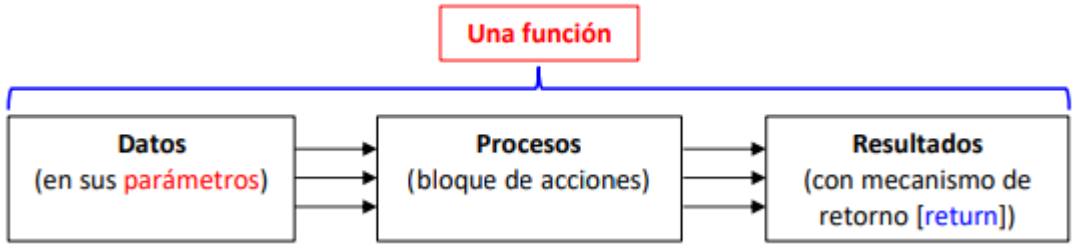
• La cabecera: también llamada encabezado de la función. Es la primera línea de la función, en ella se indica el nombre de la misma, entre otros elementos que oportunamente veremos (y que se designan como parámetros).

• El bloque de acciones: es la sección donde se indican los procesos que lleva a cabo la función. Este bloque debe comenzar a renglón seguido de la cabecera, y debe ir indentado hacia la derecha del comienzo de la cabecera. Es típico (como también veremos) que este bloque finalice con una instrucción de retorno de valor, llamada return

```
def menor(n1, n2, n3):
    if n1 < n2 and n1 < n3:
        mn = n1
    else:
        if n2 < n3:
            mn = n2
        else:
            mn = n3
    return mn
```

def menor(n1,n2,n3) es la cabecera lo que esta entre() se denominan parametros

**Figura 8: Esquema general y conceptual de una función.**





Cosas importantes a tener en cuenta a cerca de parametros en funciones:

1) Si en la cabecera de una función están declarados n parámetros formales, entonces al invocar a esa función deben enviarse n parámetros actuales.
2) El tipo de valor contenido en los parámetros actuales debería coincidir con el tipo de valor que el programador de la función esperaba para cada uno de los parámetros formales, tomados en orden de aparición de izquierda a derecha, para evitar problemas en tiempo de ejecución al intentar procesar valores de tipos incorrectos.
3) Al declarar la función debe indicarse la lista de parámetros formales de la misma, simplemente escribiendo sus nombres separados por comas. Los nombres que se designen para estos parámetros formales pueden ser cualesquiera (no importa si coinciden eventualmente o no con los nombres o identificadores de los parámetros actuales).

4) En el bloque de acciones de la función, deben usarse los parámetros formales y no los parámetros actuales. El nombre o identificador de los actuales no tiene importancia alguna dentro de la función, ya que sus valores fueron copiados en los formales. De hecho, como se dijo, los parámetros actuales podrían tener el mismo nombre que los parámetros formales, sin que esto afecte la forma de trabajo del programa ni de la función. No obstante, debe entenderse que incluso en este caso las variables actuales son diferentes de las formales (aunque se llamen igual) porque están ubicadas en lugares distintos de la memoria.No es obligatorio ni es una condicion que los parametros formales actuales tengan el mismo identificadores.

5) Si dentro del bloque de acciones de la función se altera el valor de un parámetro formal, este cambio no afectará al valor del parámetro actual asociado con él (Parametrización por copia).

Mecanismo de retorno de valores en una función en Python

Funciones con retorno de valor: devuelven un valor como resultado de la acción que realizan, de manera que ese valor puede ser usado en alguna otra operación.

| Ejemplo de uso | Explicación |
|---|---|
| `y = pow(x, n)` | El valor devuelto se asigna en una variable *y*. |
| `print(pow(x, n))` | El valor devuelto se muestra directamente en consola. |
| `y = 2 * pow(x, n)` | El valor devuelto se multiplica por dos y el resultado se asigna en *y*. |
| `y = pow(pow(x, n), 3)` | Calcula $x^n$, eleva el resultado al cubo, y asigna el resultado final en *y*. |

Funciones sin retorno de valor: realizan alguna acción pero no se espera que retornen un valor como resultado de la misma. (ejemplo : print(cadena))

Variables locales y variables globales

cualquier variable que se inicializa dentro del bloque de una función es asumida como una variable local a ese bloque (y por lo tanto, local a la función). es aquella que sólo puede usarse dentro de esa función, y no existe (o no es reconocida) fuera de ella.

Si una variable se define en el script principal (es decir, fuera de cualquier función) entonces puede ser usada en cualquier lugar del programa.

puede ser usada ya sea en el propio script principal y/o en toda función de ese programa.
se dice que esas variables son de ámbito global en el programa o que son VARIABLES GLOBALES.

FICHA 10: PROGRMACION MODULAR

Motivos por lo que se usan funciones(subrutinas)en cualquier lenguaje de programación:

El primer motivo es lograr ahorro de líneas de código, desarrollando como función a todo bloque de instrucciones que en un mismo programa se use en muchas ocasiones.
El segundo motivo es el de permitir que un programador pueda modularizar adecuadamente sus programas, dividiéndolos en subproblemas que le resulten más fáciles de manejar y controlar

la programación modular es uno de los principios básicos o estrategias de la programación estructurada

FICHA 11: MODULOS Y PAQUETES

Tratamiento especial de parámetros en Python

Parametrización: es mecanismo que favorece el desarrollo de funciones genéricas y reutilizables.

Existen básicamente tres mecanismos para lograr esto en Python, y esos tres mecanismos pueden a su vez combinarse :

*Parámetros con valores por defecto: Es común y muy útil asignar a los parámetros formales de una función un valor por defecto, de forma de poder invocar a la función enviando menos parámetros actuales. Se dice que un parámetro formal tiene un valor por defecto, cuando el mismo es asignado en forma explícita con un valor en el momento en que se define en la cabecera de la función

```
def nombre_funcion(param1, param2=valor):
    cuerpo_de_la_función
```

*Parámetros con palabra clave :  Si una función tiene parámetros asignados con valores por defecto, podría haber problemas de ambigüedad para invocarla correctamente, ya que los valores de los parámetros actuales se toman en orden de aparición para ser asignados en los formales

#en criollo de acuerdo a como yo puse los parametros en la funcion , de esa misma manera debo invocarlos en ese mismo orden.(esto no se si esta bien,se me mezclaron los conceptos)  #esta maaal mepa, pero bueno vemos despues

Ejemplo :

```
def datos(nombre, pais='Argentina', sexo='Varon', trabaja=True, estado='Soltero'):
print('Datos recibidos: ')
print('Nombre:', nombre)
print('Pais:', pais) print('Sexo:', sexo)
print('Tiene trabajo?:', trabaja)
print('Estado civil:', estado)
def test():
datos('Camila', 'Argentina', sexo='Mujer')
```

no se puede usar parametros que no existen,ni repetir parametros,luego de una palabra clave no puede seguir un explicito, ni tampoco agregar mas de un valor al mismo parametro

• Mientras se respeten las reglas anteriores, no hay problema en cambiar el orden de acceso a los parámetros usando sus palabras clave:

```
# ok... el orden no importa... si se respeta todo lo demás... datos('Bruno', sexo='Varon',
pais='Italia')
```

*Listas de parámetros de longitud arbitraria : consiste en permitir que una función acepte un número arbitrario de parámetros. Los parámetros enviados de acuerdo a esta modalidad entrarán a la función empaquetados en una tupla: esto es, entrarán a la función como una lista de valores separados por comas y accesibles por sus índices.

La forma de indicar que una función tiene una lista variable de parámetros, consiste (en general) en colocar un asterisco delante del nombre del último parámetro posicional que se prevea para la función. Ese último parámetro, definido de esta forma, representa la secuencia o tupla de valores de longitud variable.

```
def procesar_notas(nombre, nota, *args):
    # procesamiento de los parámetros normales...
    print('Notas del alumno:', nombre)
    print('Nota Final:', nota)

    # procesar la lista adicional de parámetros, si los hay
    if len(args) != 0 :
        print('Otras notas ingresadas:')
        for d in args:
            print('\tNota Parcial:', d)


def test():
    # una invocacion "normal", sin parámetros adicionales..
    procesar_notas('Carlos', 9)
    print()

    # una invocacion con tres notas adicionales...
    procesar_notas('Juan', 8, 10, 6, 7)


# script principal...…
test()
```

MODULOS
Un módulo es una colección de definiciones (variables, funciones, etc.) contenida en un archivo separado con extensión .py que puede ser importado desde un script o desde otros módulos para favorecer el reuso de código y/o el mantenimiento de un gran sistema

Para que un script pueda usar las funciones desarrolladas en un módulo externo, debe importarlo usando alguna variante de la instrucción IMPORT

*Favorece el reuso de código y/o el mantenimiento de un sistema grande.

*Los módulos en Python pueden organizarse en carpetas llamadas paquetes (o package).
.
#VARIANTES DE IMPORT:

from- import: Esta variante incluye el nombre de las función específica que se quiere acceder, o varias de ellas separadas por comas, evitando usar el operador punto.

from soporte import menor ---> solo uso la funcion menor

from soporte import * ----> Se pueden usar “todas” funciones del módulo soporte

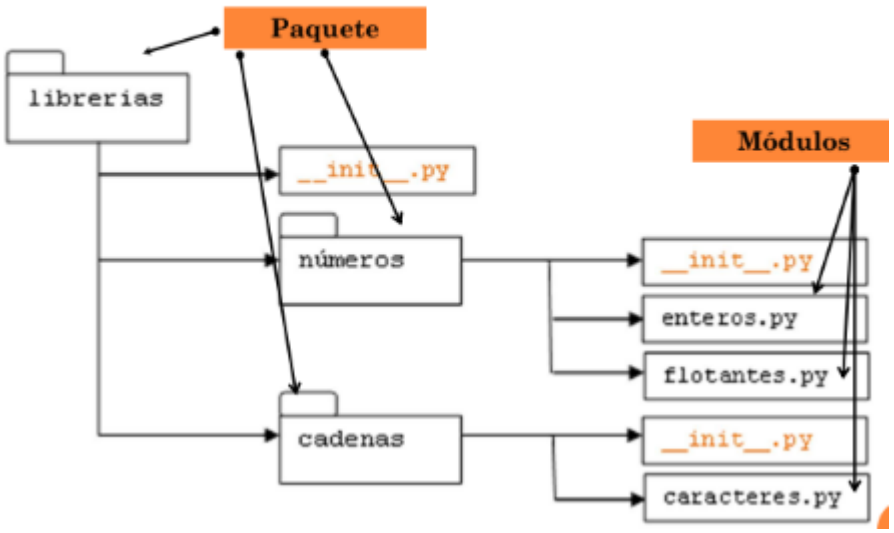
#LIBRERIA ESTANDAR EN PYTHON

| Módulo estándar | Contenido general |
|---|---|
| **os** | Interfaz de acceso a funciones del sistema operativo. |
| **glob** | Provee un par de funciones para crear listas de archivos a partir de búsquedas realizadas con caracteres "comodines" (wildcards) |
| **sys** | Variables de entorno del shell, y acceso a parámetros de línea de órdenes. También provee elementos para redireccionar la entrada estándar, la salida estándar y la salida de errores estándar. |
| **re** | Herramientas para el reconocimiento de patrones (expresiones regulares) en el procesamiento de strings. |
| **math** | Funciones matemáticas típicas (logarítmicas, trigonométricas, etc.) |
| **random** | Funciones para el trabajo con números aleatorios y decisiones aleatorias. |
| **urllib.request** | Funciones para recuperación de datos desde un url. |
| **smtplib, poplib** | Funciones para envío y recepción de mails. |
| **email** | Gestión integral de mensajes de email, permitiendo decodificación de estructuras complejas de mensajes (incluyendo attachments). |
| **datetime** | Clases para manipulación de fechas y horas. |
| **zlib, gzip, bz2, zipfile, tarfile** | Los cinco módulos citados contienen funciones para realizar compresión/descompresión de datos, en diversos formatos. |
| **timeit, profile, psstats** | Proveen funciones para realizar medición de rendimiento de un programa, sistema, bloque de código o instrucción. |
| **doctest** | Herramienta para realizar validaciones de tests que hayan sido incluidos en strings de documentación. |
| **unittest** | Similar a doctest, aunque no tan sencillo. Permite realizar tests más descriptivos, con la capacidad de poder mantenerlos en archivos separados. |
| **xmlrpc.client, xmlrpc.server** | Permiten realizar llamadas a procedimientos remotos, como si fuesen tareas triviales. |
| **xml.dom, xml.sax** | Soporte para el parsing de documentos XML. |
| **csv** | Lectura y escritura en formato común de "comma separated values" (CSV). |
| **gettext, locale, codecs** | Soporte de "*internationalization*". |

#PAQUETES

es una forma de organizar y estructurar carpetas de módulos de forma que luego se pueda acceder a cada módulo mediante el conocido recurso del operador punto para especificar la ruta de acceso y el nombre de cada módulo Así, por ejemplo, si se tiene un paquete (o sea, una carpeta de módulos) llamado modulos, y dentro de él se incluye un módulo llamado funciones, entonces el nombre completo del módulo sería modulos.funciones y con ese nombre deberá ser accedido desde una instrucción import.

## DEFINICIÓN DE PAQUETES EN PYTHON.




__name__ es una variable global del módulo que se asigna con “__main__”. Se puede conocer si alguien pide ejecutar el módulo.

__all__ los elementos a importar desde un paquete

__doc__ todos los docstrings que contiene el elemento

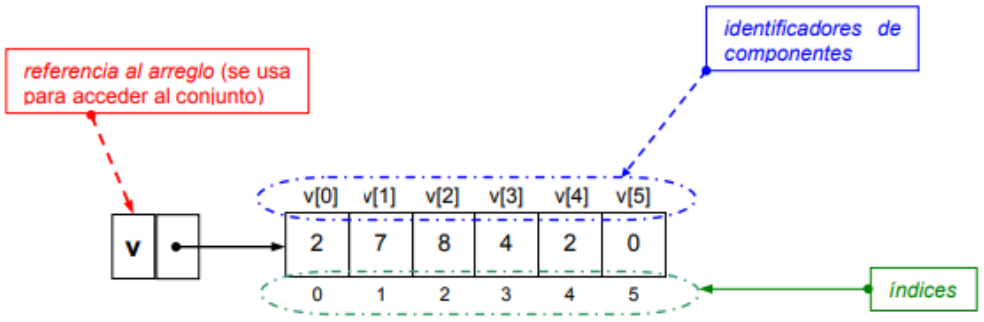
__autor__ el nombre del autor del elemento

FICHA 12 : ARREGLOS UNIDIMENSIONALES

Es una colección de valores que se organiza de tal forma que cada valor o componente individual es identificado automáticamente por un número designado como índice. El uso de los índices permite el acceso, uso y/o modificación de cada componente en forma individual.

La cantidad de indices representa la dimension del arreglo.

**Figura 1: Esquema básico de un arreglo unidimensional.**




Una variable tipo list() son una secuencia mutable.

2 maneras de representar un arreglo o vector vacio:

v = [ ]
v = list()

como es una secuencia mutable podemos modificar de varias maneras sus valores:

```
# asigna el valor 4 en la casilla 3
v[3] = 4

# suma 1 al valor contenido en la casilla 1
v[1] += 1

# muestra en consola el valor de la casilla 0
print(v[0])

# asigna en la casilla 4 el valor de la casilla 1 menos el de la 0
v[4] = v[1] - v[0]

# carga por teclado un entero y lo asigna en la casilla 5
v[5] = int(input('Valor: '))

# resta 8 al valor de la casilla 2
v[2] = v[2] - 8
```

Si el arreglo se creó vacío y luego se desea agregar valores al mismo, debe usarse el método append provisto por el propio tipo list, ejemplo :

notas = [ ]
notas.append(4)
notas.append(7)

print('Notas:', notas) # muestra: Notas: [4, 7]

como ya se que si necesito crear un vector con n componenetes iguales o n componenetes que no se la cantidad, se procede a hacer lo siguiente :

```
ceros = 15 * [0]
print('Arreglo de 15 ceros:', ceros)    # muestra: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Para crear arreglos a partir de otras  secuencias (cadena de caracteres,tuplas, etc) se usa la func.constructora list():

```
x = 'Mundo'
letras = list(x)
print(letras)
# muestra: ['M', 'u', 'n', 'd', 'o']
```

Para cargar un vector por teclado:

```
notas = []

for i in range(4):
    x = int(input('Nota: '))
    notas.append(x)

print('Notas:', notas)
```

ACCESO A COMPONENTES INDIVIDUALES DE UNA VARIABLE TIPO LIST.

las variable de tipo list que se usan para representar arreglos no son de tamaño fijo, sino que pueden aumentar o disminuir en forma dinámica su tamaño, agregar o quitar elementos en cualquier posición o incluso cambiar el tipo de sus componentes .

Puedo eliminiar un elemento en particular o tambien eliminar el vec complero:

```
numeros = [3, 5, 8, 6, 9, 2]
del numeros[3] #si o si entre corchetes por que estamos usando vectores
print(numeros)      # muestra: [3, 5, 8, 9, 2]
```

Tambien puedo  acceder a un subrango de casilleros ‘rebanando’ o ‘cortando’ sus índices :

```
’numeros = [4, 2, 3, 4, 7]
nueva = numeros[1:4]  #toma las casillas del 1 al 4 y el 4 viene a ser como range llegas hasta el 3
print("\nNueva lista: ", nueva)                # muestra: [2, 3, 4]
```

Asi como puedo ‘acortar’ las list tambien puedo sumarle algo a ese rango de numeros, o multiplicarlos , modificar sus valores etc.

UN EJEMPLO DE TANTOS:

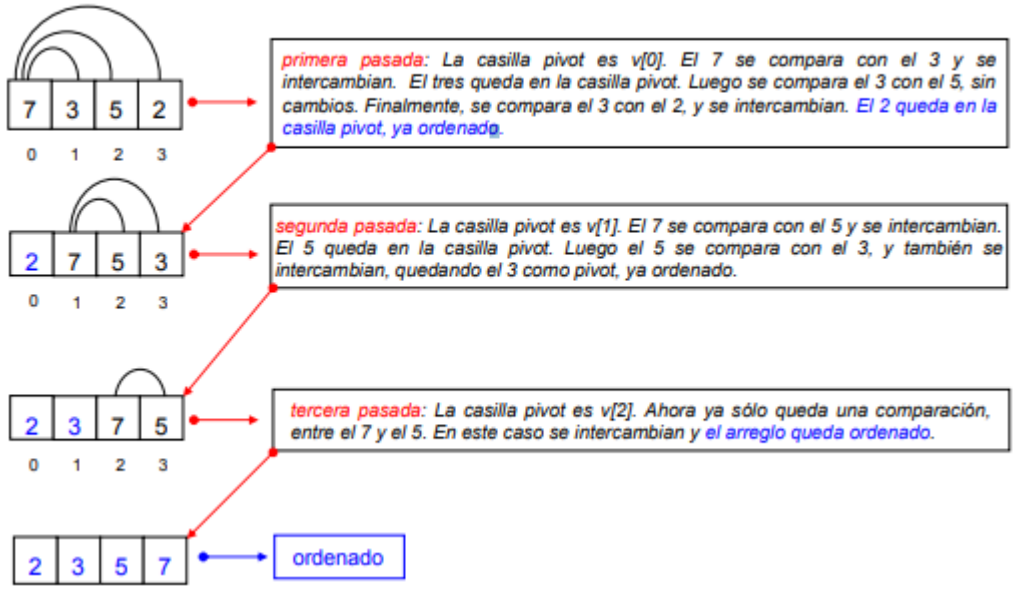
```
lis = [47, 17, 14, 41, 37, 74, 48]
lis[2:5] = [10, 20, 30]
print(lis)
# muestra: [47, 17, 10, 20, 30, 74, 48]
```

TODO VECTORES SE RECORRE CON CICLO FOR
FICHA 13 :TECNICAS BASICAS

ORDENAMIENTO DE UN ARREGLO(SELECCIÓN DIRECTA)

```
def selection_sort(v):
# ordenamiento por seleccion directa
        n = len(v)
        for i in range(n-1):
                for j in range(i+1, n):
                        if v [ i ] > v[ j ]:
                                v[ i ], v[ j ] = v[ j ], v[ i ].
```

**Figura 1: Esquema de trabajo completo del algoritmo de ordenamiento por Selección Directa.**




BUSQUEDA SECUENCIAL.

Este metodo es el que te dice en que posicion esta si lo encuentra y si no te informa un mensaje que no se encontro.

```
def linear_search(v, x):
 # busqueda secuencial...
        for i in range(len(v)):
                if x == v[i]:
```

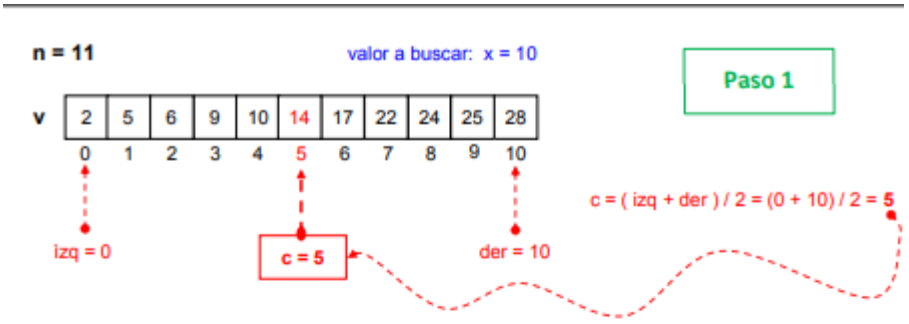
```
                return i
        return -1
```

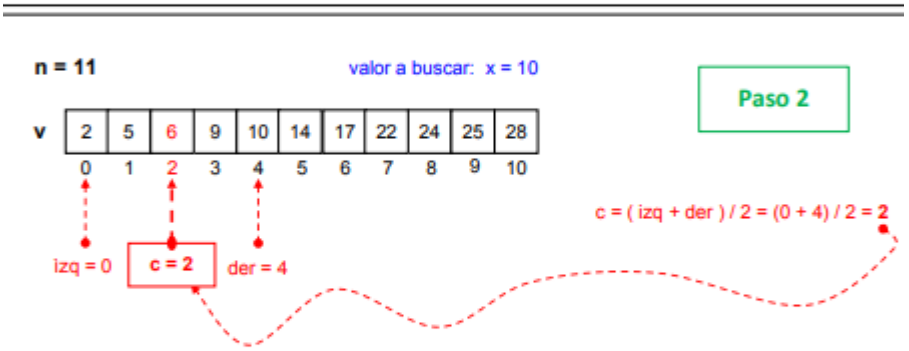
BUSQUEDA BIANARIA

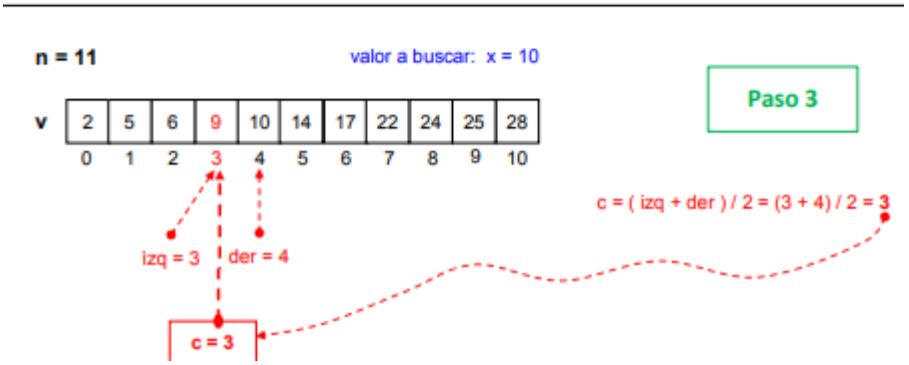
Este se utiliza cuando el arreglo esta ordenado

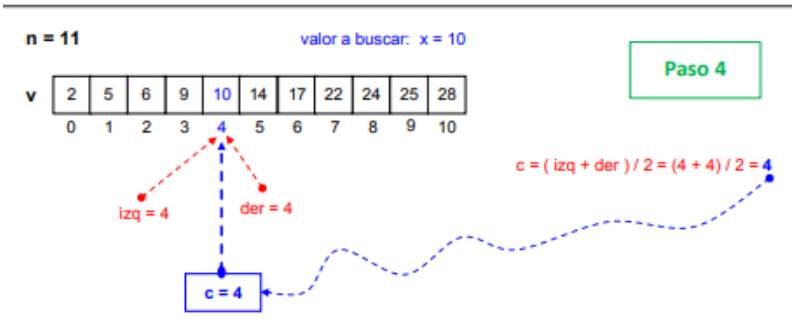
```
def binary_search(v, x):
        # busqueda binaria... asume arreglo ordenado...
        izq, der = 0, len(v) - 1
        while izq <= der:
                c = (izq + der) // 2
                if x == v[c]:
                        return c
                if x < v[c]:
                        der =   c - 1
                else:
                        izq = c + 1
        return -1
```

esquema de como trabaja la busqueda binaria




Una desventaja obvia es que el arreglo debe sí o sí estar ordenado para utilizar este algoritmo.

FICHA 14 : ARREGLOS - CASOS DE ESTUDIOS I

ARREGLOS PARALELOS

Se utilizan cuando es necesario almacenar información en varios arreglos unidimensionales a la vez, pero de tal forma que haya correspondencia entre los valores almacenados en la casillas con el mismo índice (llamadas casillas homólogas).

**Figura 1: Dos arreglos *correspondientes* o *paralelos*.**

| | | | | | |
|---|---|---|---|---|---|
| *nombres* | Juan | Ana | Alejandro | María | Pedro |
| *sueldos* | 1100 | 2100 | 1300 | 750 | 800 |
| | **0** | **1** | **2** | **3** | **4** |

• El manejo de ambos arreglos es simple: sólo debe recordar el programador que hay que usar el mismo índice en ambos arreglos para entrar a la información de una misma persona

```
print('Nombre:', nombres[2])     # Alejandro
print('Sueldo:', sueldos[2])     # 1300
```

ejemplo

CONTEO Y ACUMULACION POR ACCESO DIRECTO(VEC CONTEO Y ACUMULACION)

Se utilizan cuando se requiere llevar a cabo un conteo masivo.
Un vector de conteo es un arreglo en donde cada uno de sus componentes se comporta como un contador.

En un en un vector de acumulación, cada componente se comporta como un acumulador

**Contador**

```
con = con + 1
vec[ indice ] = vec[ indice ] + 1
```

**Acumulador**

```
acu = acu + var
vec[ indice ] = vec[ indice ] + var
```

FICHA 15 : ARREGLOS - CASOS DE ESTUDIOS II

Hay muchos ejericicios de arreglos.

ENCRIPTACIÓN SIMPLE DE MENSAJES

Cifrado de Transposición

Esta técnica OBTIENE la letra o símbolo de reemplazo, a través de una segunda tabla (que llamaremos tabla de transposición) la cual puede estar formada por los mismos símbolos del alfabeto original (¡pero en otro orden!) o bien puede estar formada directamente por otros símbolos.

| **Alfabeto:** | A | B | C | D | E | F | G | H | I | J | K | L | M | N | Ñ | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Transposición:** | R | I | S | Q | P | A | N | O | W | X | U | M | D | H | Z | T | F | G | B | L | E | Y | K | C | J | V | Ñ |

FICHA 16 : ARREGLOS BIDIMENSIONALES(MATRICES)

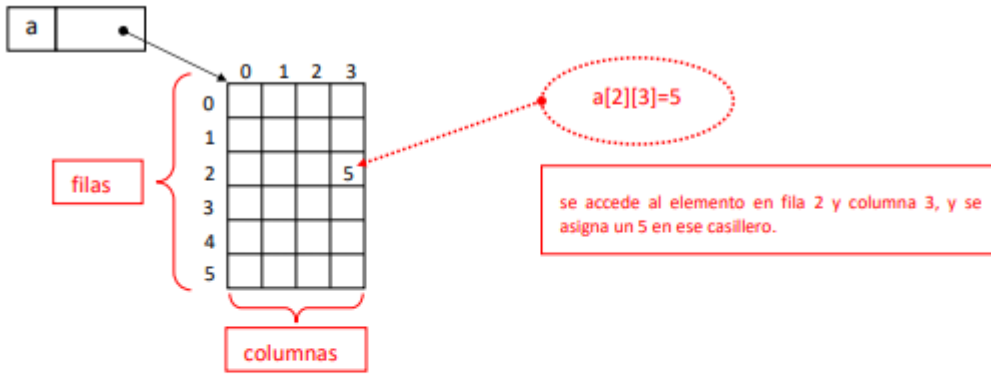
CREACION Y USOS DE LOS ARREGLOS BIDIMENSIONALES

Un matriz es un arreglo cuyos elementos están dispuestos en forma de tabla, con varias filas y columnas. Aquí llamamos filas a las disposiciones horizontales del arreglo, y columnas a las disposiciones verticales.

formas de represnetar matrices simples :

```
m0 = [ [1, 3, 4],
       [3, 5, 2],
       [4, 7, 1] ]
       # se puede indentar aqui o en la columna del corchete de apertura...

print('Matriz con valores fijos:', m0)

m0 = [ [1, 3, 4], [3, 5, 2], [4, 7, 1] ]
print('Matriz con valores fijos:', m0)   #es lo mismo , la primera es para entender mejor.
```

**Figura 1: Esquema de representación de una matriz de orden 6*4.**




```
a = [ [2, 3, 4, 5],
      [1, 5, 6, 3],
      [7, 2, 4, 0],
      [8, 4, 1, 7],
      [0, 0, 2, 9],
      [1, 3, 4, 1],
    ]

a[2][3] = 5  # accede a una casilla y cambia su valor.
```

formas de representar una matriz nxm:

-1

```
m1 = [ ]
for f in range(n):
    m1.append([ ])
    for c in range(m):
        m1[f].append(None)
```

La matriz asi construida tendra el sig aspecto con n= 3 y m =4:
m1 --->[ [None, None, None, None], [None, None, None, None], [None, None, None, None] ]

-forma 2 por comprension :

m3 = [ [None] * m for f in range(n) ]

m1[0][3] = 10 #para acceder a un elemento 2 indices uso, primero el de la fila y el segundo el de la columna

Para recorrer por filas de forma completa

```
# un recorrido por filas
for f in range(len(m2)):
    for c in range(len(m2[f])):
        m2[f][c] = f * c
```

Recorre la matriz por columna:

```
# recorrido por columnas – matriz regular
filas = 3 columnas = 4
for c in range(columnas):
        for f in range(filas):
                m3[f][c] = f * c
```

FICHA 17 : ORDENAMIENTO

**Figura 1: Clasificación tradicional de los algoritmos de ordenamiento más comunes.**

| Algoritmos Simples o Directos | Algoritmos Compuestos o Mejorados |
|---|---|
| Intercambio Directo (Burbuja) | Método de Ordenamiento Rápido (*Quicksort*) [C. Hoare - 1960] |
| Selección Directa | Ordenamiento de Montículo (*Heapsort*) [J. Williams – 1964] |
| Inserción Directa | Ordenamiento por Incrementos Decrecientes (*Shellsort*) [D. Shell – 1959] |

Clasificación tradicional:

Algoritmos simples o directos: Basados en ideas intuitivas muy simples.

Algoritmos compuestos o mejorados: Se fundamentan en estrategias lógicas muy sutiles y no tan obvias a primera vista.

La diferencia no es solo conceptual, sino que existe una diferencia de rendimiento muy marcada en cuanto el tiempo que demoran.

FUNCIONAMIENTO DE ALGORITMOS DE ORDENAMIENTOS SIMPLES O DIRECTOS

ORDENAMIENTO POR INTERCAMBIO DIRECTO(BUBBLESORT):
Es el mas lento de todos

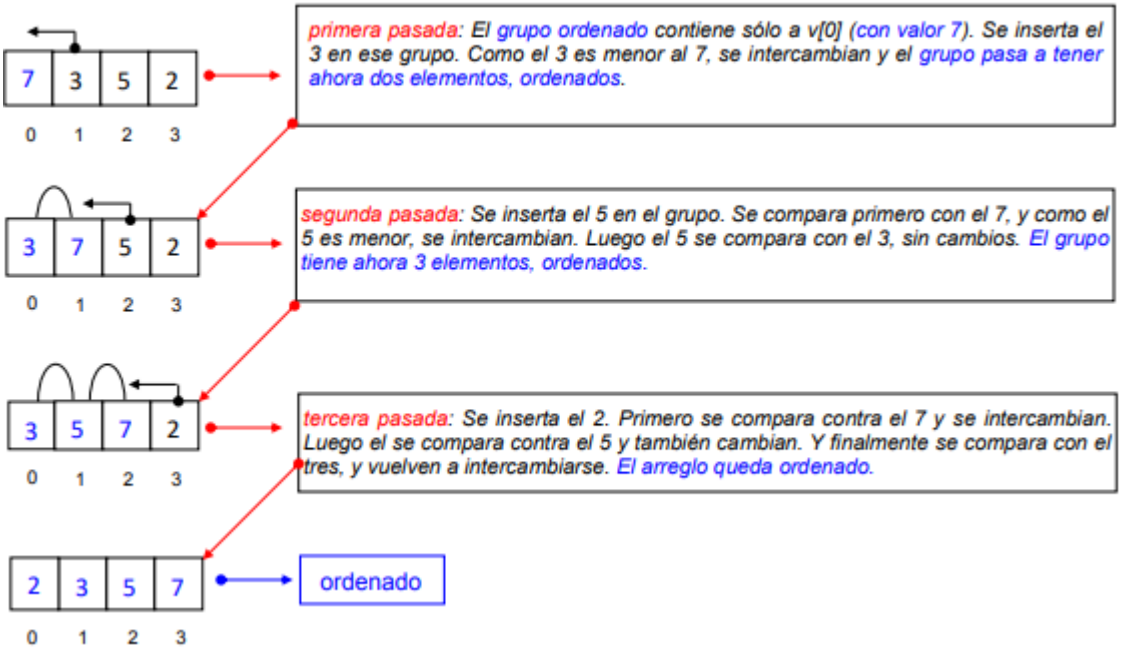
el objetivo es que cada elemento en cada casilla v[i] se compara con el elemento en v[i+1]. Si este último es menor, se intercambian los contenidos. Se usan dos ciclos anidados para conseguir que incluso los elementos pequeños ubicados muy atrás, puedan en algún momento llegar a sus posiciones al frente del arreglo

```
def bubble_sort(v):
        n = len(v)
        for i in range(n-1):
                ordenado = True
                for j in range(n-i-1):
                        if v[j] > v[j+1]:
                                ordenado = False
                                v[j], v[j+1] = v[j+1], v[j]
                if ordenado:
                    break
```

ORDENAMIENTO POR INSERCION DIRECTA(O INSERCIÓN SIMPLE):

Se comienza suponiendo que el valor en la casilla v[0] conforma un subconjunto. Y como tiene un solo elemento, está ordenado. A ese subconjunto lo llamamos un grupo ordenado dentro del arreglo.  Se toma v[1] y se trata de insertar su valor en el grupo ordenado. Si es menor que v[0] se intercambian, y si no, se dejan como estaban.  Se toma v[2] y se procede igual, comenzando la comparación contra v[1] (que es el mayor del grupo)

**Figura 3: Funcionamiento general del Algoritmo de *Ordenamiento por Inserción Directa*.**



```
def insertion_sort(v):
        n = len(v)
        for j in range(1, n):
                y = v[j] .
                k = j - 1
                while k >= 0 and y < v[k]:
                        v[k+1] = v[k]
                        k -= 1
                v[k+1] = y
```

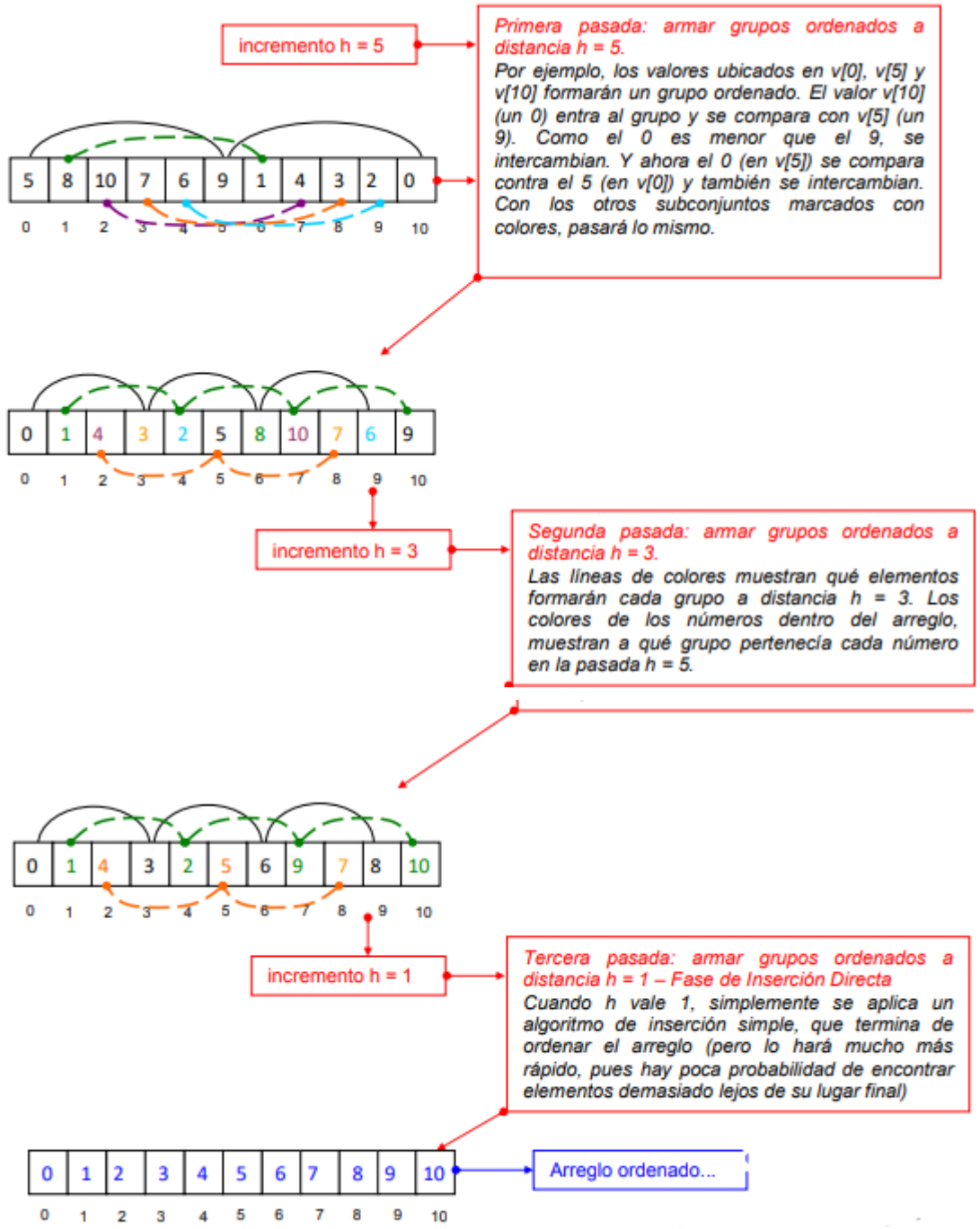
ALGORITMOS DE ORDENAMIENTOS COMPUESTOS O MEJORADOS

ORDENAMIENTO DE SHELL(SHELLSORT)

Es una mejora del ordenamiento por inserción directa.  Compara los elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada

La idea es lanzar un proceso de ordenamiento por inserción, pero en lugar de hacer que cada valor que entra al grupo se compare con su vecino inmediato a la izquierda, se comience haciendo primero un reacomodamiento de forma que cada elemento del arreglo se compare contra elementos ubicados más lejos, a distancias mayores que uno, y se intercambien elementos a esas distancias.  Luego, en pasadas sucesivas, las distancias de comparación se van acortando y repitiendo el proceso con elementos cada vez más cercanos unos a otros.  De esta forma, se van armando grupos ordenados pero no a distancia uno, sino a distancia h tal que h > 1.
Finalmente, se termina tomando una distancia de comparación igual a uno, y en ese momento el algoritmo se convierte lisa y llanamente en una Inserción Directa para terminar de ordenar el arreglo.  Las distancias de comparación se denominan en general incrementos decrecientes, y de allí el nombre con que también se conoce al métodO

**Figura 4: Esquema general de funcionamiento del Algoritmo *Shellsort*.**



```
def shell_sort(v):
    n = len(v)
    h = 1
    while h <= n // 9:
        h = 3*h + 1
        while h > 0:
            for j in range(h, n):
                y = v[j]
                k = j - h
                while k >= 0 and y < v[k]:
                        v[k+h] = v[k]
                        k -= h
                v[k+h] = y
            h //= 3
```

EL ALGORITMO HEAPSORT

**Figura 5: Funcionamiento general del Algoritmo *Heapsort* - Fase 1: Armado del Grupo Inicial.**

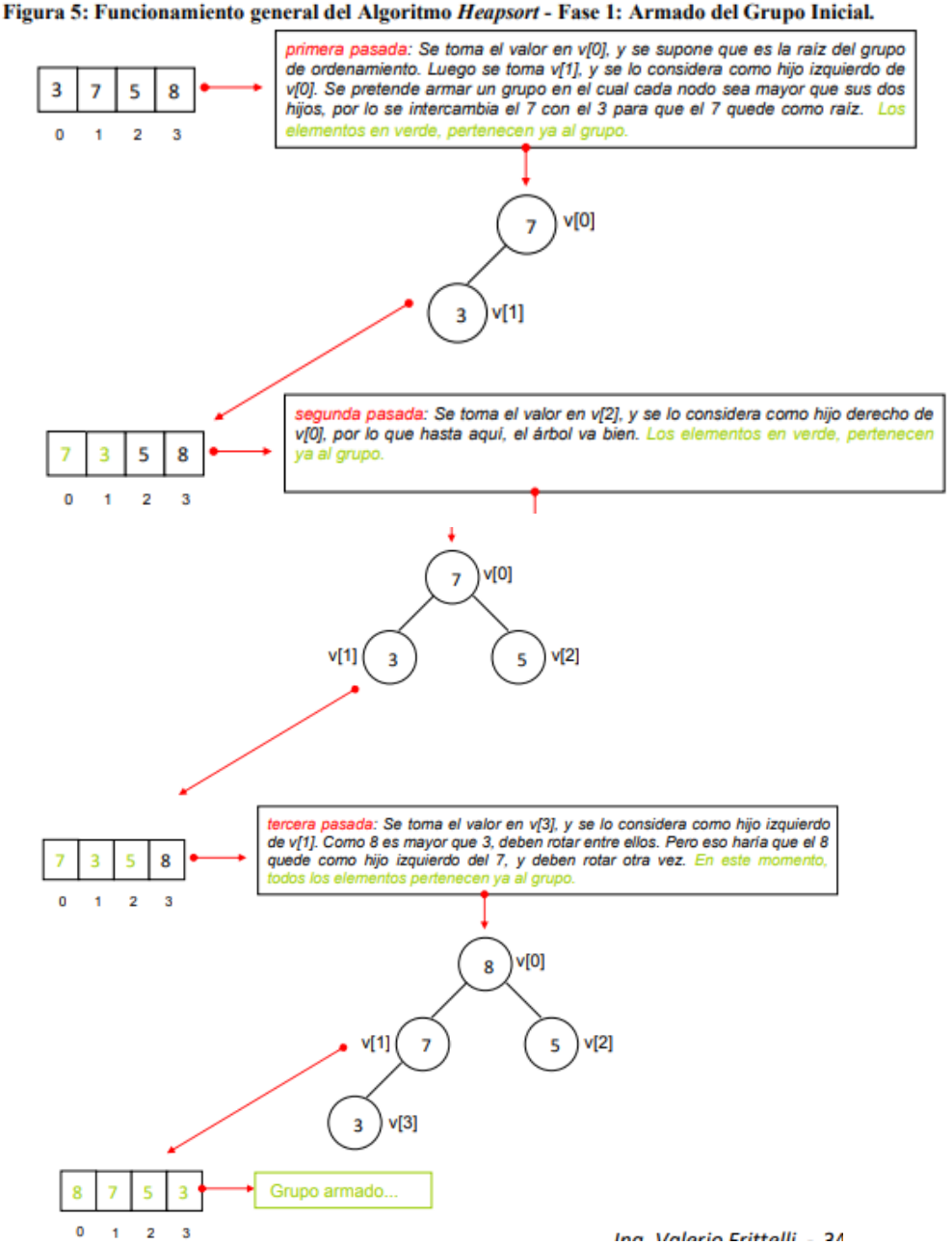
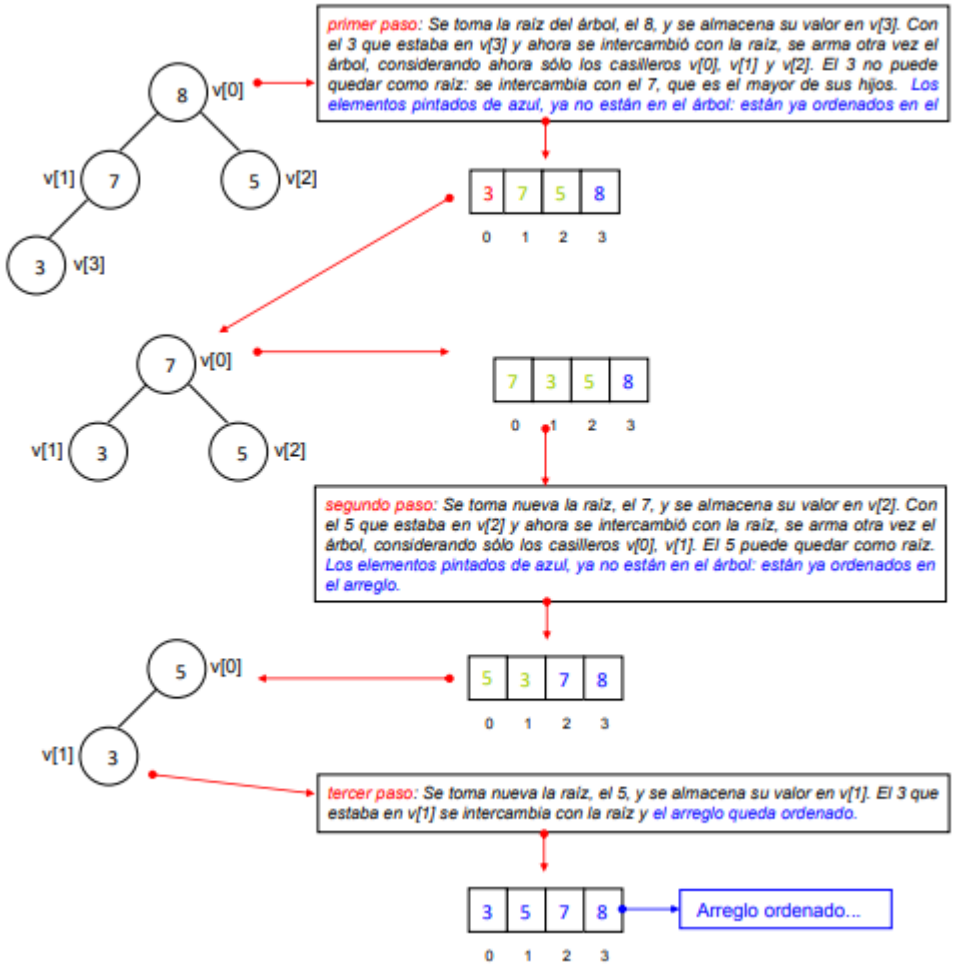


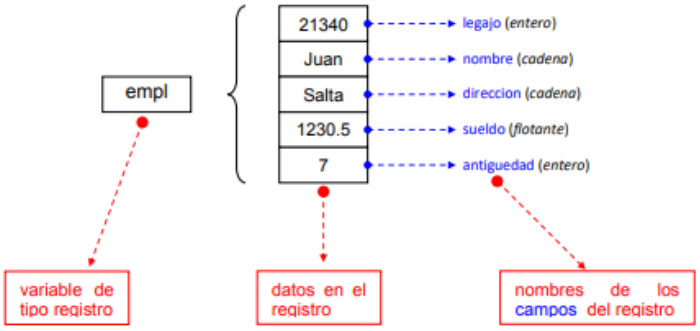
**Figura 6: Funcionamiento general del Algoritmo *Heapsort* - Fase 2: Ordenamiento Final.**




FICHA 18 : REGISTROS

Un registro es un conjunto mutable de valores que pueden ser de distintos tipos. Cada componente de un registro se denomina campo (o también atributo, dependiendo del contexto).
Los elementos individuales de un registro se acceden por medio de un *nombre* o *identificador* declarado por el programador, en lugar de hacerlo por medio de índices

**Figura 1: Esquema de una variable de tipo registro.**




en este ejemplo un registro llamado emple, contiene 5 campo que son legajo,nombre,direccion ,sueldo y antigüedad

para definir en python un registro , uso la palabra reservada CLASS

para acceder a un campo del registro, uso el '.' y el nombre del campo

```python
# declaración de un tipo registro vacio
class Empleado:
    pass


# una función para inicializar un registro de tipo Empleado
def init(empleado, leg, nom, direc, suel, ant):
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant


# una función para mostrar un registro de tipo Empleado
def write(empleado):
    print("\nLegajo:", empleado.legajo, end=' ')
    print("- Nombre:", empleado.nombre, end=' ')
    print("- Direccion:", empleado.direccion, end=' ')
    print("- Sueldo:", empleado.sueldo, end=' ')
    print("- Antiguedad:", empleado.antiguedad, end=' ')


# una funcion de prueba
def test():
    # creación de variables vacias de tipo Empleado...
    e1 = Empleado()
    e2 = Empleado()
    e3 = Empleado()

    # inicializacion de campos de las tres variables...
    init(e1, 1, 'Juan', 'Calle 1', 10000, 10)
    init(e2, 2, 'Luis', 'Calle 2', 20000, 15)
    init(e3, 3, 'Pedro', 'Calle 3', 25000, 20)
```

```python
    # visualizacion de los valores de los tres registros..
    write(e1)
    write(e2)
    write(e3)


# script principal...
if __name__ == '__main__':
    test()
```

FICHA 19 : PILAS Y COLAS

La que siempre utilizamos son estructuras nativas que son las que nos provee python.
Estas son estructuras abstractas  por que son estructuras ‘diseñadas’  por el programador

IMPLEMENTACION DE ESTRUCTURAS ABSTRACTAS

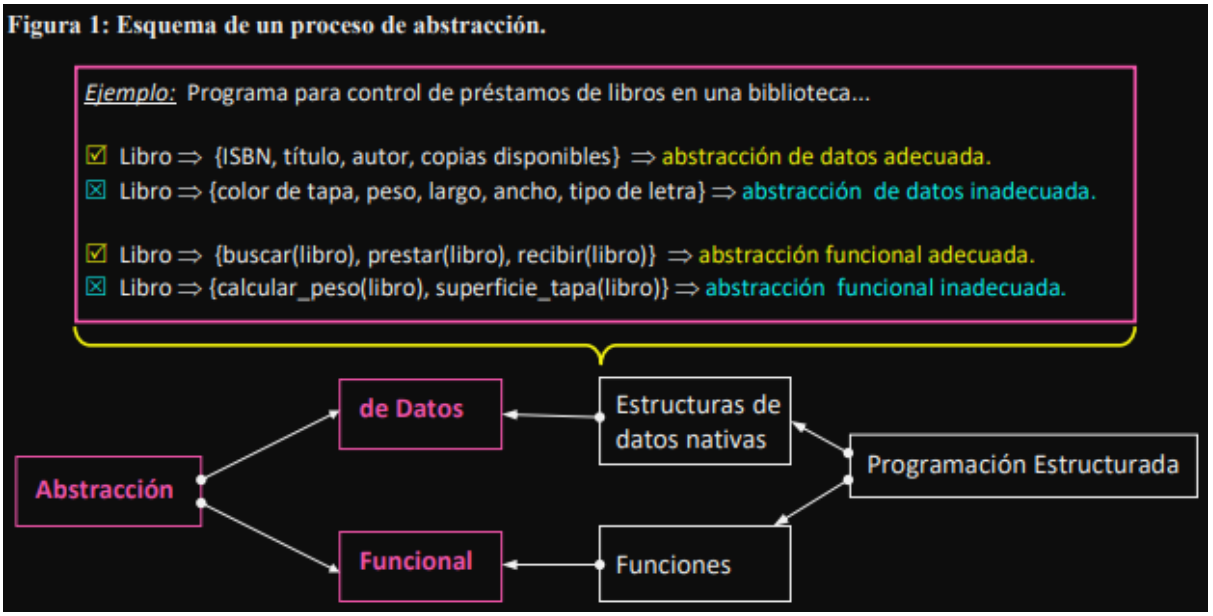
el proceso de implementación es muy importante ,debo darme cuenta cuando debo usar EA

El proceso llevado a cabo por el programador para crear el nuevo tipo/estructura abstracto (Libro en este ejemplo), se conoce con el nombre general de implementación del tipo abstracto.

*Lo primero es aplicar un mecanismo conceptual denominado Mecanismo de abstracción : es el proceso por el cual se intenta captar y aislar la información y el comportamiento esencial de una entidad u objeto del dominio (enunciado) del problema

*abstracción es también una simplificación de la realidad.

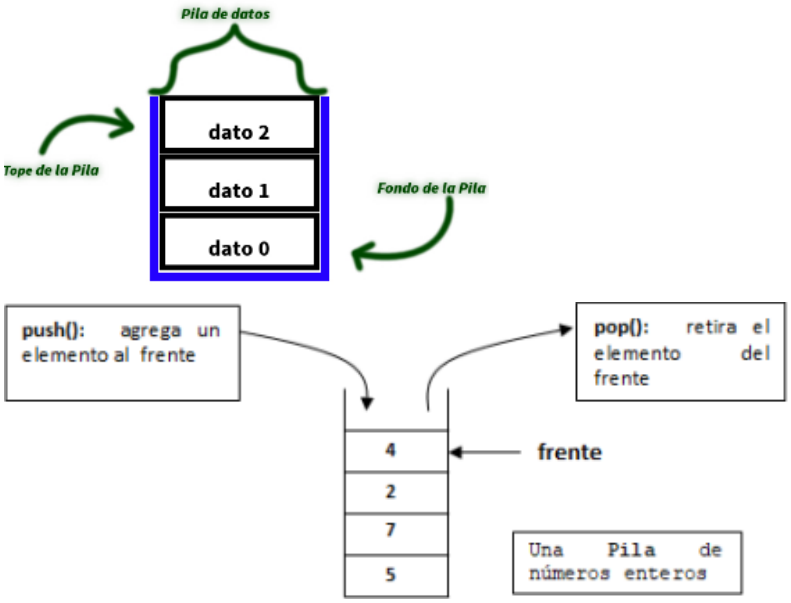
el mecanismo de abstraccion consta de dos partes: captar y aislar los datos relevantes (o abstracción de datos); e identificar los procesos aplicables sobre esos datos (lo que se designa como abstracción funcional).
Por relevantes, se entiende a los datos y procesos que realmente son necesarios en el programa de acuerdo al dominio o enunciado.
Cuando aquí se habla de datos relevantes y de procesos relevantes se quiere expresar que el programador debe hacer un análisis detallado de cuáles son los datos que realmente necesita en su programa para representar a una entidad del enunciado o dominio del problema, y cuales son los procesos o algoritmos que realmente necesita implementar en su programa para procesar esos datos.




PILAS

Es una estructura lineal (cada elemento tiene un unico sucesor y un unico antecesor), los elementos se organizan en la forma de la que van ingresando , y para volcar o sacar esos elementos el ultimo es el primero que sale y asi sucesivamente , y obvio el primero que entro es el ultimo en salir.

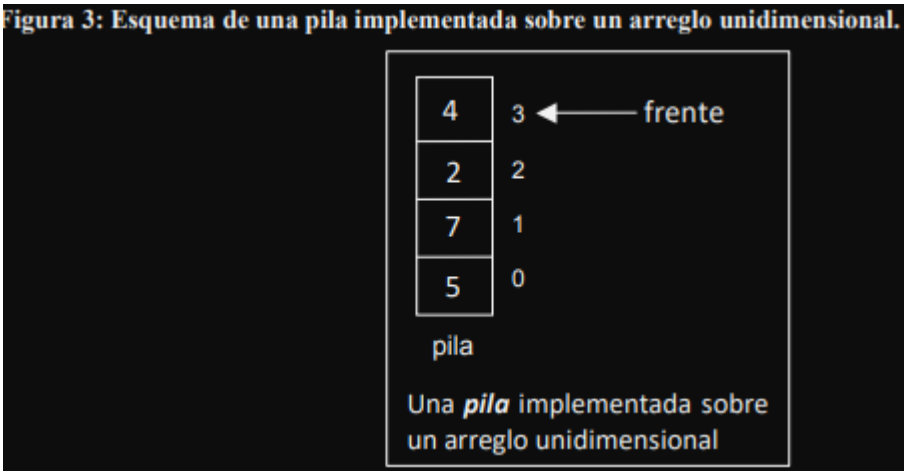
ejemplo de como es una pila.




IMPLEMENTACION DE LA PILA

Abstracción de datos: Lo típico es usar un vector para simular la propia pila. El fondo de la pila se puede hacer coincidir con el casillero cero del vector, y el elemento del frente estará entonces en la última casilla.

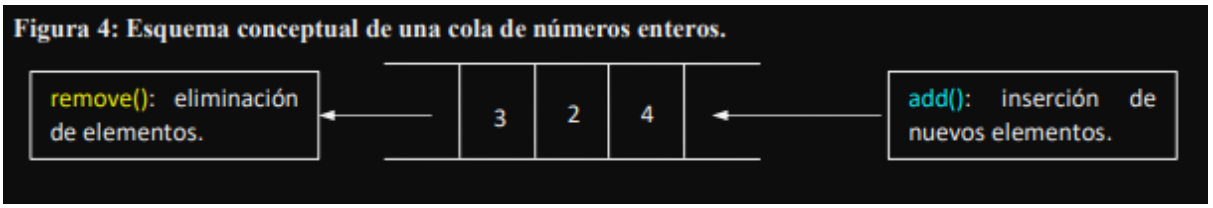
Abstracción funcional: La operación de inserción de un nuevo elemento puede hacerse con un función push(), y la eliminación con otra función llamada pop() (entre otras operaciones...)




COLAS

Una cola es una estructura lineal en la cual los elementos se organizan uno detrás del otro, quedando uno de ellos al principio (o al frente) de la cola y otro en el último lugar de la misma (o al fondo).
.
*este es lo contrario a pilas, el primero que entra sale.()FIFO, First in - First out)

para agregar un elemento usamos add() y para eliminar un elemento remove()




...

FICHA 22 : ARCHIVOS

Son conjuntos de datos persistentes (es decir, no volátiles): un archivo es una estructura de datos que en lugar de almacenarse en memoria principal, se almacena en dispositivos de almacenamiento externos (como discos o memorias flash) y por lo tanto el contenido de un archivo no se pierde al finalizar el programa que lo utiliza

al utilizar archivos puedo manejar grandes volumenes de archivos
los datos que graban en los archivos , se representan en sistema bianario.

tenemos dos tipos de archivos:

- Archivos de texto:Todos los bytes del archivo son interpretados como caracteres.Si un archivo es de texto también contiene bytes, pero se asume que todos esos bytes representan caracteres que pueden ser visualizados en pantalla, con la única excepción del caracter de salto de línea.("\n").Pero desde el caracter 32 (que es el espacio en blanco) en adelante, todos tienen representación visual (por ejemplo, el caracter 65 es la letra "A").Para visualizar en pantalla lo que contiene el archivo de texto, debe contener solo los bytes cuyos valores numericos sean mayores o iguales a 32, obvio von la excepcion de el salto de linea.

- Archivos binarios:Las secuencias o bloques de bytes que el archivo contiene representan información de cualquier tipo (números en formato binario, caracteres, valores booleanos, etc.) y no se asume que cada byte representa un caracter.
  a este tipo de archivos no los puedo leer por pantalla, por que provoca una secuencia de datos inilegible,

para abrir un archivo solo basta invocar a la funcion open()

m = open('datos.dat','w')  # aca solo llamar a open y el primer parametro es el nombre } que yo desee ponerle al archivo, y el segundo parametro es el modo de apertura.

m = open("c:\\documents\\datos.dat", "w")  #Si no se indica la ruta de acceso, el intérprete Python lo buscará en la carpeta actual del
proyecto o programa que se está ejecutando

**Tabla 1: Distintos modos de apertura de un archivo en Python.**

| Modo | Significado |
|---|---|
| **r** (o **rt**) | El archivo se abre como *archivo de texto* en *modo de solo lectura* (no está permitido grabar). No será creado en caso de no existir previamente. Este es el modo por defecto si se invoca a *open()* sin especificar modo de apertura alguno. |
| **w** (o **wt**) | El archivo se abre como archivo *de texto* en *modo de solo grabación*. Si ya existía, su contenido se perderá y se abrirá vacío. Si el archivo no existía, será creado. |
| **a** (o **at**) | El archivo se abre como archivo *de texto* en *modo de solo append* (todas las grabaciones se hacen al final del archivo, *preservando* su contenido previo si el archivo ya existía). Si no existía, será creado. |
| **r+** (o **r+t**) | El archivo se abre como *archivo de texto* en *modo de lectura y grabación*. El archivo debe existir previamente: no será creado en caso de no existir. |
| **w+** (o **w+t**) | El archivo se abre como archivo *de texto* en *modo de grabación y lectura*. Si ya existía su contenido será eliminado y abierto vacío. Si no existía, será creado. |
| **a+** (o **a+t**) | El archivo se abre como archivo *de texto* en modo de *lectura y de append* (todas las *grabaciones* se hacen al final del archivo, preservando su contenido previo). Si no existía, será creado. |
| **rb** | El archivo se abre como *archivo binario* en *modo de sólo lectura*. No será creado en caso de no existir previamente. |
| **wb** | El archivo se abre como archivo *binario* en *modo de sólo grabación*. Si ya existía su contenido será eliminado. Si no existía, será creado. |
| **ab** | El archivo se abre como archivo *binario* en *modo de sólo append* (todas las grabaciones se hacen al final del archivo, preservando su contenido previo si el archivo ya existía). Si no existía, será creado. |
| **r+b** | El archivo se abre como *archivo binario* en *modo de lectura y grabación*. El archivo debe existir previamente: no será creado en caso de no existir. |
| **w+b** | El archivo se abre como archivo *binario* en *modo de grabación y lectura*. Si ya existía su contenido será eliminado. Si no existía, será creado. |
| **a+b** | El archivo se abre como archivo *binario* en *modo de lectura y de append* (todas las *grabaciones* se hacen al final del archivo, preservando su contenido previo si ya existía). Si no existía, será creado. |

open abre el canal de comunicacion, asi como es super importante abrirlo al canal de comunicación tambien es importante cerrarlo,para eso solo llamo a CLOSE()

Los objetos de tipo file (file object) que se crean con open() contienen numerosos métodos adicionales para el manejo del archivo. Entre ellos, existen los métodos read() y write() que permiten respectivamente leer y grabar datos en el archivo [2]. Ambos métodos son directos y simples de usar cuando se trata de archivos de texto, pero no son tan directos ni tan simples cuando se trata de leer o grabar en un archivo binario, sobre todo si la intención es trabajar con registros.

La serialización es un proceso por el cual el contenido de una variable normalmente de estructura compleja (como puede ser un registro, un objeto, una lista, etc.) se convierte automáticamente en una secuencia de bytes listos para ser almacenados en un archivo, pero de tal forma que luego esa secuencia de bytes puede recuperarse desde el archivo y volver a crear con ella la estructura de datos original [2] [3]. El mecanismo de serialización no sólo está disponible en Python sino también en muchos otros lenguajes (sobre todo orientados a objetos, Java entre ellos).

En nuestro caso, emplearemos el módulo pickle, que entre otras funciones, provee las dos que necesitaremos: dump() y load().

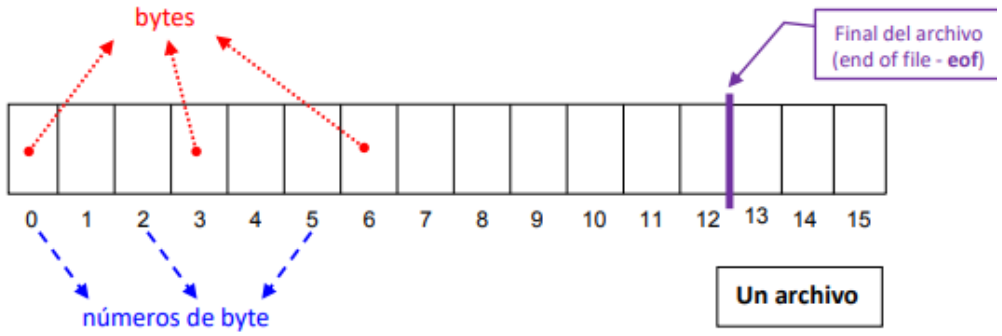
ejemplo de como funcionan los modulos pickle y load.

```
import os.path
import pickle

__author__ = 'Cátedra de AED'


def test():
    print('Procediendo a grabar números en el archivo')
    m = open('prueba.num', 'wb')
    x, y = 2.345, 19
    pickle.dump(x, m)
    pickle.dump(y, m)
    m.close()

    m = open('prueba.num', 'rb')
    a = pickle.load(m)
    b = pickle.load(m)
    m.close()
    print('Datos recuperados desde el archivo:', a, ' - ', b)

    print('Hecho...')


if __name__ == '__main__':
    test()
```

ACCESO DIRECTO Y RECORRIDO SECUENCIAL

un archivo (binario o de texto) puede entenderse como un gran arreglo o vector de bytes ubicado en memoria externa en lugar de estar alojado en memoria principal. Cada componente de ese gran arreglo en memoria externa es uno de los bytes grabados en el archivo, y cada byte tiene (a modo de índice) un número que lo identifica, correlativo desde el cero en adelante. El byte cero es el primer byte del archivo. Notar que entonces el número de cada byte es un indicador de su posición relativa al inicio del archivo.

**Figura 1: Esquema conceptual de un archivo como un arreglo de bytes en memoria externa.**




En muchas situaciones el programador necesita obtener el tamaño en bytes de un archivo. Hay varias formas de hacerlo en Python, pero el siguiente ejemplo muestra la que quizás sea la más simple, usando la función getsize() incluida en el módulo os.path .

```
t = os.path.getsize('libros.dat')
print(t)
```

os.path.getsize() toma como parámetro el nombre físico del archivo (que aquí suponemos que se llama 'libros.dat', y simplemente retorna su tamaño en bytes. Note que no es necesario que el archivo esté abierto con open() para poder usar esta función.

todo archivo cuenta con una especie de cursor o indicador o marcador interno llamado file pointer, que no es otra cosa que una variable de tipo entero tal que, mientras el archivo está abierto, contiene el número del byte sobre el cual se realizará la próxima operación de lectura o de grabación. Tanto las operaciones de lectura como las de grabación, comienzan la operación en el byte indicado por el file pointer y al terminar la misma, dejan el file pointer apuntando al byte siguiente a aquel en el cual terminó la operación. Si una operación de salida graba bytes detrás del byte de finalización del archivo, entonces el tamaño del archivo crece y se ajusta para abarcar los nuevos bytes

**Figura 2: Esquema del estado del *file pointer* en un archivo antes de proceder a una lectura.**

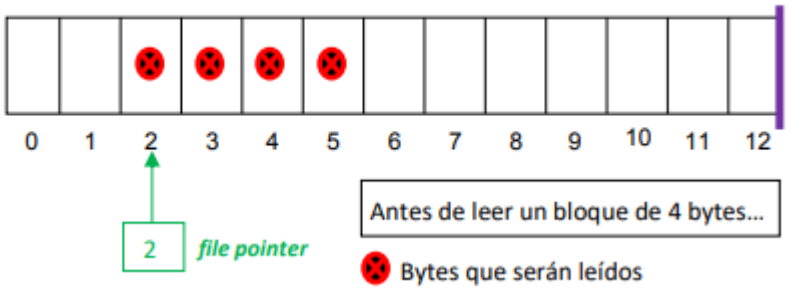
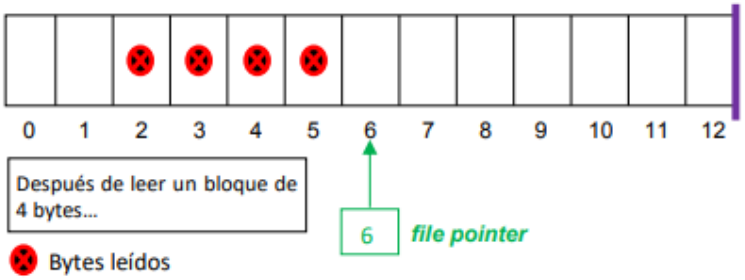


**Figura 3: Esquema del estado del *file pointer* en un archivo después de proceder a una lectura.**




El acceso y recorrido en forma secuencial es común y simple de implementar. Muchos procesos requieren justamente tomar todos los datos de un archivo, uno por uno en el orden en que aparecen, y procesarlos en ese orden. Por ejemplo, esto será necesario si se desea hacer un listado de todo el contenido del archivo, o buscar un dato particular en ese archivo

En ese sentido, los objetos tipo file object creados con open() contienen un método tell() que retorna el valor del file pointer en forma de un número entero, y también un método seek() que permite cambiar el valor del mismo (y por lo tanto, seek() permite elegir cuál será el próximo byte que será leído o grabado

a función tell() es muy útil cuando por algún motivo se requiere saber en qué byte está posicionado un archivo en un momento dado. Esto es particularmente necesario cuando se quiere leer el contenido completo de un archivo en forma secuencial, usando un ciclo. Suponga que se tiene un archivo que contiene numerosos registros grabados mediante pickle.dump() y se necesita leer ese archivo y mostrar su contenido en pantalla. Está claro que el programador requerirá un ciclo, de forma de hacer una lectura (con pickle.load()) y una visualización en cada iteración

Por su parte, el método seek(offset, from_what) permite cambiar el valor del file pointer. En general recibe dos parámetros: el primero (que aquí llamamos offset) indica cuántos bytes debe moverse el file pointer, y el segundo (llamado aquí from_what, si está presente) indica desde donde se hace ese salto (un valor 0 indica saltar desde el principio del archivo, un 1 desde donde está el file pointer en este momento y un 2 indica saltar desde el final). El valor por default del segundo parámetro es 0, por lo cual por omisión se asume que los saltos son desde el inicio del archivo. El valor del segundo parámetro puede ser indicado como un número o por medio de una de las siguientes tres constantes predefinidas (las tres dentro del módulo io que debe ser importado en el programa para poder usarlas)

**Tabla 2: Constantes predefinidas para el método seek().**

| Constante | Valor | Significado |
|---|---|---|
| io.SEEK_SET | 0 | Reposicionar comenzando desde el principio del archivo. El valor del primer parámetro (offset) puede ser 0 o positivo (pero no negativo). |
| io.SEEK_CUR | 1 | Reposicionar comenzando desde la posición actual del puntero de registro activo. El valor de offset puede ser entonces negativo, 0 o positivo. |
| io.SEEK_END | 2 | Reposicionar comenzando desde el final del archivo. El valor de offset típicamente es negativo, aunque puede ser 0 o positivo. |

Si no se indica el segundo parámetro al invocar a seek(), se asume por defecto que su valor es 0, y en ese caso el reposicionamiento se hará desde el inicio del archivo

En general, la forma de entender el funcionamiento de este método consiste en suponer que el file pointer está apuntando al lugar indicado por el segundo parámetro, y sumar a esa posición el valor del primer parámetro, veamos algunos ejemplos:

a)Sin importar donde esté ubicado el file pointer en este momento, suponga que queremos cambiar su valor para que pase a valer 10. Podemos hacerlo así:

```
m.seek(10, io.SEEK_SET)
```

Si el segundo parámetro es SEEK_SET (o sea, 0), se debe asumir que el file pointer está ubicado al inicio del archivo (o sea, su valor sería el 0), a ese valor sumarle el 10 que se pasó como primer parámetro, y asignar el resultado en el file pointer que pasará entonces a valer el número 10

b) Sin importar donde esté ubicado el file pointer en este momento, suponga que queremos cambiar su valor para que salte al final del archivo. Podemos hacerlo así:

```
m.seek(0, io.SEEK_END)
```

Si el segundo parámetro es SEEK_END (o sea, 2), se debe asumir que el file pointer está ubicado al final del archivo (o sea, su valor sería el número del primer byte que está fuera del archivo), a ese valor sumarle el 0 que se pasó como primer parámetro, y asignar el resultado al file pointer que pasará entonces a valer el número del primer byte que está fuera del archivo (es decir, el file pointer quedará apuntando al final)

c) Suponga que el file pointer en este momento está apuntando al byte 7 del archivo (el valor del file pointer es 7), y suponga que queremos cambiar su valor para que pase a apuntar al byte 4. Podemos hacerlo así:

```
m.seek(-3, io.SEEK_CUR)
```

Si el segundo parámetro es SEEK_CUR (o sea, 1), se debe asumir que el file pointer está ubicado en su posición actual (o sea, su valor sería el byte 7 que ya tenía), a ese valor sumarle el -3 que se pasó como primer parámetro, y asignar el resultado en el file pointer. Como 7 + (-3) = 4, el file pointer pasará a valer 4.

d) Está claro que el ejemplo anterior también podría haber sido resuelto así:

```
m.seek(4, io.SEEK_SET)
```

Si el segundo parámetro es SEEK_SET (o sea, 0), se debe asumir que el file pointer está ubicado al inicio del archivo (o sea, su valor sería el 0), a ese valor sumarle el 4 que se pasó como primer parámetro, y asignar el resultado en el file pointer, que pasará a valer 4.

FICHA 24 : ARCHIVOS - APLICACIONES
hay ejercicios -y la explicacion del add in order

FICHA 25: ARCHIVOS DE TEXTO

Se usa casi siempre ascii, cada caracter se representa exactamente con un byte, dejando el primer bit en 0 y usando el resto de los siete bits para representar el número de orden del caracter. Como sólo se usan siete bits, se pueden representar entonces no más de 128 caracteres.

A medida que paso el tiempo el conjunto de 128 caracteres empezo a ser insuficiente , Una de esas variantes se conoce como ASCII 8 y emplea los ocho bits de un byte para representar caracteres, con lo cual el número total se extiende a 256 caracteres (y es la variante que se usó para incorporar caracteres especiales no incluidos en el idioma inglés pero comunes en otros idiomas (como los de base latina): la ñ del español, la ç del francés o el portugués, o las letras vocales acentuadas

y al ver que habia idiomas que no se podian representar  con los ascii, se planteó el estándar Unicode con fundamentos más complejos pero mucho más amplio que el estándar ASCII. Esencialmente, Unicode admite como un subconjunto válido al estándar ASCII, por lo que un texto codificado en ASCII será perfectamente legible por una aplicación o programa basado en Unicode.

A su vez, se han planteado tres formas de codificación de caracteres en base al estándar Unicode, designadas como UTF-8, UTF-16 y UTF-32 (y siendo UTF la abreviatura de Unicode Trasformation Format o Formato de Transformación Unicode), y a su vez estas tres formas dan lugar a numerosos esquemas técnicos de codificación
En general, es suficiente con saber que mediante Unicode (en cualquiera de las tres formas de codificación que se emplee) se puede representar la totalidad de los caracteres existentes en todos los idiomas, incluidas lenguas muertas, lenguas orientales, lenguas latinas y no latinas y caracteres gráficos especiales.

Python por default emplea el Unicode UTF-8 para interpretacion de textos

USOS DE ARCHIVOS DE TEXTO DE PYTHON

se utiliza de la misma forma que con archivos bianarios.

```
m = open('datos.txt', 'w+t')
```

LOS MODOS  DE APERTURA EN LA TABLA DE LA FICHA 22.

Una vez que el archivo ha sido abierto, la variable file object (en nuestro caso, m) que representa al archivo dispone de varios métodos (tales como write(), read(), readline() y readlines()) que permiten grabar o leer cadenas de caracteres con sencillez desde ese archivo.

| Método | Acción |
|---|---|
| m.write(cad) | Graba la cadena *cad* en el archivo, **sin** agregar un salto de línea al final de la misma (puede agregarlo el programador). |
| cad = m.read() | Lee y retorna el contenido **completo** del archivo, como una **única** cadena. |
| cad = m.readline() | Lee una **cadena simple** desde el archivo. Comienza en la posición del file pointer y termina al encontrar un salto de línea o el final del archivo. Mantiene el salto de línea en la cadena retornada. |
| cad = m.readlines() | Lee **todas** las líneas del archivo, y retorna una **lista** o **arreglo** con todas ellas (una por cada casilla del arreglo). Mantiene los saltos de línea al final de cada cadena leída. |

Reposicionamiento del file pointer en un archivo de texto en Python

Como sabemos, el método tell() retorna un número entero con el valor que en ese momento tenga el file pointer del archivo, medido en la cantidad de bytes desde el inicio del mismo y considerando al primer byte como en la posición 0(cero). Recuerde que si el file pointer está ubicado al final de un archivo (el primer byte inmediatamente luego del último que pertenece al archivo), entonces el valor retornado por tell() indica el tamaño en bytes del archivo (al igual que lo que ya vimos para archivos binarios) :

```
m = open('prueba', 'w')
m.write('Universidad')
pos = m.tell()
# muestra: 11
print('Posicion del file pointer / Tamaño del archivo:', pos)
```

No confundir tamaño del archivo en bytes, con cantidad de caracteres del archivo. En el formato Unicode – esquema UTF-8, un caracter puede llegar a ser representado con 1, 2, 3 o 4 bytes dependiendo de cuál sea ese caracter, por lo que no necesariamente un tamaño de k bytes es equivalente a k caracteres en un archivo de texto

El ya conocido método seek() permite cambiar el valor del file pointer también en un archivo de texto, aunque ahora existen algunas restricciones menores que distinguen su uso respecto de los archivos binarios.

Vimos que seek() en general recibe dos parámetros: el primero indica cuántos bytes debe moverse el file pointer, y el segundo indica desde donde se hace ese salto (el valor io.SEEK_SET = 0 indica saltar desde el principio del archivo, el valor io.SEEK_CUR = 1

indica saltar desde donde está el file pointer en ese momento y el valor io.SEEK_END = 2 indica saltar desde el final). El valor por default del segundo parámetro es io.SEEK_SET = 0, por lo cual por omisión se asume que los saltos son desde el inicio del archivo (revise la Ficha 22 para repasar estos conceptos si lo necesita)

Otros metodos.

| Método o Atributo | Aplicación |
|---|---|
| closed | Es un atributo o campo (no un método) cuyo valor es *True* si el archivo está cerrado. |
| flush() | Vuelca al archivo los buffers de grabación, si corresponde. Aplicable sólo para archivos abiertos para grabar (no hace nada en caso contrario). |
| readable() | Retorna True si el archivo está disponible para ser leído. |
| truncate(size=None) | Trunca el contenido del archivo a una cantidad igual a *size* bytes. El archivo puede *aumentar* o *disminuir* su tamaño. |
| writable() | Retorna *True* si el archivo está disponible para ser grabado. |
| writelines(lines) | Graba el contenido de la lista *lines* en el archivo, por defecto ***sin*** incluir separadores de línea (por lo cual, el programador debe indicar ese separador al final de cada elemento de la lista *lines* si quiere los separadores). |

Procesamiento de archivos de texto que contienen secuencias numéricas

Muchas veces se tienen archivos de texto que contienen series de números expresados como cadenas Es común que en estos casos cada línea contenga un "número" y el salto de línea que separa a un número del siguiente. Cuando se necesita procesar un archivo así, la idea es leer las cadenas que representan números en el archivo, convertirlas a números en el formato correcto (int o float).
Guardar números como cadenas en un archivo, se hace (por ejemplo) para evitar problemas de incompatibilidad de formatos numéricos, El formato de texto también tiene diversas variantes, pero es siempre más sencillo compatibilizar formatos de texto que numéricos, sobre todo si el archivo de texto se genera usando un juego de caracteres estándar (ASCII, UTF-8, etc.)

FICHA 23: ARCHIVOS : GESTION ABM

TRABAJA CON REGISTROS

Gestion abm: operaciones que se realizan sobre registros

*Agregar un registro al archivo: alta de un registro.
*Eliminar un registro: baja de un registro.
* Modificar Modificar el contenido contenido de un registro registro que ya existía: modificación de un registro.

BAJAS EN UN ARCHIVO DE REGISTROS

Pasos generales:
*Buscar un registro con esa clave en el archivo:
*En caso de encontrarlo, encontrarlo, proceder proceder a eliminar eliminar el registro completo
*En Python no existe una instrucción que permita eliminar un registro del archivo:

se usan 2 archivos uno temporal(wb) y el original(rb) , recorro con un ciclo for, y se lee con pickle.load() hasta el final del original, en cada vuelta el ciclo se toma el registro leído desde archivo original y se verifica si el campo clave coincide con el valor buscado. Si el valor no coincide, entonces simplemente se toma ese registro y se lo graba (con pickle.dump()) en el archivo temporal. El único registro que no se copia al temporal, es aquel cuyo campo clave coincida con el valor buscado

*Soluciones clásicas:

*baja fisica:

Grabar en un archivo temporal, inicialmente vacío, los datos del archivo original sin el registro que se quiere eliminar (aquel cuyo campo clave coincida con el valor original Borrar el original Renombrar el temporal con el nombre del original temporal eliminar (aquel cuyo campo clave coincida con el valor buscado). Al finalizar el proceso, el archivo temporal será una copia exacta del archivo original, salvo que no contendrá al registro que se quería borrar. Se procede a destruir el archivo original (borrarlo del sistema de archivos o file system del disco), y cambiar el nombre del archivo temporal para que ahora tome el nombre que tenía el original.

*baja logica:

en vez de eliminarlo físicamente, simplemente se lo marca usando un campo especialmente definido a modo de bandera: si este campo vale False, significa que el registro está marcado como eliminado y por lo tanto no debe tenerse en cuenta en ninguna operación que se realice sobre el archivo, pero si vale True, entonces el registro está activo y por lo tanto debe ser procesado cuando se recorra el archivo.Consiste en buscar el registro con la clave dada usando un ciclo, leerlo con pickle.load(), cambiar el campo de marcado lógico a False, y volver a grabar el registro en el mismo lugar original usando pickle.dump().

*combinación de ambas:

Mejor opcion

Plantear una solución de equilibrio entre las ventajas de ambas estrategias, se suele aplicar un esquema combinado:
Baja lógica: Para hacer bajas en tiempo real (por ejemplo, en casos de atención frente al público).
Baja Física: Se efectúa el proceso en algún momento no crítico de ciclo de uso del archivo (por ejemplo, a la hora del cierre o de la apertura de la organización que usa el sistema). Se elimina de una sola pasada todos los registros que quedaron marcados. Esto se llama depuración del archivo (o compactación u optimización del espacio físico del archivo)

ALTAS

Altas de registros con claves repetidas

*Ingresar los datos por teclado del nuevo registro, se asigna en True su campo de marcado marcado lógico, y se graba el registro al final del archivo.

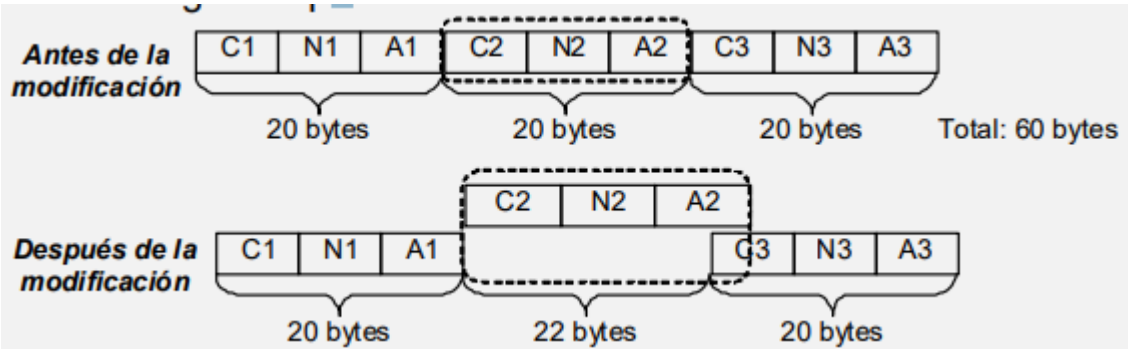
*Ingresar la clave de identificación del registro que se quiere agregar (por ejemplo el campo legajo).
*Buscar un registro registro con esa clave en el archivo archivo y que además no esté marcado como eliminado
*Si se encuentra, entonces la operación de alta se rechaza
*Si no se encontró repetida la clave, entonces se terminan de cargar los datos por teclado, se asigna en True su campo de marcado lógico, y se graba el registro al final del archivo.

MODIFICACIONES Y LISTADOS

Ingresar la clave de identificación del registro que se quiere modificar (por ejemplo el campo legajo). Buscar un registro con esa clave en el archivo y que además no esté marcado marcado como eliminado eliminado. Si se encuentra, se muestran los campos del mismo por pantalla y con un pequeño menú de opciones se pide al operador que elija los campos que quiere modificar, cargando por teclado los valores nuevos. Para finalizar se vuelve a grabar el registro cambiado en la posición original

Para que estas técnicas funcionen los registros del archivo deben ser del mismo tamaño, es decir, deben tener la misma cantidad de bytes

Atención:  Si esto no se controla, por ejemplo, cuando se quiera modificar modificar un registro registro de un tamaño determinado determinado por otro de tamaño mayor o menor, se estará modificando la secuencia de registros posterior.




En Python una variable ocupará tantos bytes como sea necesario para poder representar en binario el valor asignado asignado en ella, por lo tanto esa asignación asignación es dinámica y no la controla el desarrollador.  Solución: El desarrollador tendrá que validar los datos para mantener el tamaño de los registros.

Cadena de caracteres: prefijar el tamaño de misma, usando la función ljust() (fuerza a la cadena a tener tantos caracteres como indica el número tomado como parámetro) y usar algún método para truncar la cadena si se pasa del tamaño prefijado:

```
cadena.ljust(30,' ')[:30]
```

LISTADOS

Numeros enteros : Validar el valor del número entero para asegurar que siempre tenga un valor en un intervalo conocido, y asegurarse que todos los números en ese intervalo tengan el mismo tamaño en bytes

Numero float :  Siempre se representan con doble precisión (8 bytes) y cuando un float es serializado mediante pickle.dump() su representación se expande a 12 bytes en el archivo.

Listado completo : Muestra el contenido del archivo completo:
Se recorre el archivo en forma secuencial y se muestran en pantalla todos los registros que no estén marcados como eliminados.

Listado parcial : (listado con filtro)

Se recorre el archivo en forma secuencial y se lee todo el archivo, pero sólo se muestran en pantalla los registros que cumplen con cierta condición (por ejemplo, mostrar sólo los registros de los alumnos cuyo promedio sea mayor o igual a 7)
BUSQUEDAS

Buscar la posición (file pointer) de un registro en particular.

Pasos generales:
*Ingresar la clave de identificación del registro que se quiere buscar(por buscar(por ejemplo ejemplo el campo legajo).
*Buscar un registro con esa clave en el archivo.
*Si no se encuentra retornar una posición inválida, por ejemplo -1 o None.
*Si se encuentra, detener la búsqueda y retornar el número del byte interno del archivo (file pointer) en el cual comienza el registro que se acaba de encontrar.

FICHA 20 y 21 : ANALISIS DE ALGORITMOS

El Análisis de Algoritmos es la rama de las Ciencias de la Computación que se orienta a técnicas para medir la eficiencia de un algoritmo.
Esa medición de eficiencia o rendimiento, se hace con relación a algún factor o parámetro medible. Los dos factores que típicamente se miden, son el tiempo de ejecución y el consumo de memoria, aunque el primero de ellos suele ser el más comúnmente aplicado.
A su vez, el análisis debe tener en cuenta situaciones que favorecen totalmente a un algoritmo (el "mejor caso"), o que lo perjudican por completo (el "peor caso"). La primera lleva a un planteo optimista (poco práctico en general) y la segunda a un contexto pesimista (que puede ser poco probable). Por eso muchas veces se prefiere el análisis del "caso promedio" (no favorable ni desfavorable, pero más complejo de realizar)

La meta es plantear fórmulas que permitan estimar de alguna forma el tiempo de demora o la cantidad de memoria empleada, y NO simplemente hacer pruebas y cronometrar tiempos o medir consumo de bytes: la fórmula es general, mientras que la medición puntual refleja lo que ocurrió en esa prueba, bajo esas circunstancias. Nos centraremos de aqui en más en el factor tiempo.
Para el planteo de esas fórmulas, se toma como entrada la cantidad de datos n (o tamaño del problema) que el algoritmo debe procesar, y se intenta detectar la operación crítica: aquella que por repetirse muchas veces, hace que el algoritmo demore lo que demora.

se intenta deducir qué fórmula mide mejor la cantidad de veces que se ejecuta la operación crítica para valores de n cada vez mayores. Y en lo que resta de esta explicación supondremos un análisis del peor caso (pesimista pero más simple de plantear).

Ejemplo: Ordenamiento por Selección Simple

En el algoritmo de ordenamiento por selección simple, el tamaño del problema es el tamaño n del vector. Si la operación crítica es la comparación de dos elementos, entonces siempre cae en el peor caso, ya que siempre hace todas las comparaciones posibles (el intercambio de valores también podría ser tomado como crítico, pero nos centraremos en las comparaciones para mantener simple el análisis

Queremos una fórmula que nos diga cuántas comparaciones hace el algoritmo para distintos valores de n, y como dada esa fórmula, el tiempo se deduce indirectamente, con esa fórmula será suficiente.

El algoritmo hace n-1 pasadas. En la primera, hace n-1 comparaciones, en la segunda hace n-2, y sigue así hasta la última pasada en la que sólo hace una comparación. Por lo tanto, la cantidad total de comparaciones es t(n) = (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1. Pero se puede probar que eso equivale a t(n) = ((n-1) * n) / 2, o bien: t(n) = ½ n 2 - ½ n

Conteo Exhaustivo vs Análisis Asintótico

La fórmula anterior fue el resultado de un análisis de conteo riguroso o exhaustivo de operaciones críticas. Pero en muchos casos, para un programador es suficiente un análisis más general, que sólo permita poner de manifiesto cuál es el comportamiento de la función para valores grandes de n (¿es cuadrática? ¿es lineal? ¿es exponencial?...) Eso se conoce como análisis asintótico: se busca captar la forma general de variación de la función, y para ello, se aplica una notación conocida como notación "O mayúscula" o "Big O": se expresa la función, pero desprovista de constantes y términos que no son dominantes cuando n tiende al infinito. En nuestro caso, si se toma la fórmula rigurosa t(n) = ½ n 2 - ½ n y se la lleva a notación Big O, quedaría:

Notacion de big O: funciones de aparcion frecuente

**Tabla 1: Funciones típicas en el análisis de algoritmos.**

| Función | Significado (cuando mide tiempo de ejecución) | Casos típicos |
|---|---|---|
| **O(1)** | Orden constante. El tiempo de ejecución es constante, sin importar si crece el volumen de datos. | Acceso directo a un componente de un arreglo. |
| **O(log(n))** | Orden logarítmico. Surge típicamente en algoritmos que dividen sucesivamente por dos un lote de datos, desechando una parte y procesando la otra. | Búsqueda binaria. |
| **O(n)** | Orden lineal. Se da cuando cada uno de los datos debe ser procesado una vez. | Búsqueda secuencial.<br>Recorrido completo de un arreglo. |
| **O(n*log(n))** | Surge típicamente en algoritmos que dividen el lote de datos, procesando cada partición sin desechar ninguna, y combinando los resultados al final. No hemos analizado aún algoritmos que respondan a este orden. | Ordenamiento Rápido (Quick Sort). |
| **$O(n^2)$** | Orden cuadrático. Típico de algoritmos que combinan dos ciclos de *n* vueltas cada uno. | Ordenamiento por Selección Directa. |
| **$O(n^3)$** | Orden cúbico. Típico de algoritmos que combinan tres ciclos de *n* repeticiones cada uno. No hemos analizado aún algoritmos que respondan a ese orden | Multiplicación de matrices. |
| **$O(2^n)$** | Orden exponencial. Algoritmos que deben explorar una por una todas las posibles combinaciones de soluciones cuando el número de soluciones crece en forma exponencial. | Problema del viajante. Solución recursiva de la Sucesión de Fibonacci. |

FORMAS DE CRECIEMIENTO.

Las funciones que vimos (y cualquier otra en notación Big O) se conocen como ***funciones de orden de complejidad***, y para n que tiene al infinito crecen de forma tal que:

$$O(1) < O(\log(n)) < O(n) < O(n*\log(n)) < O(n^2) < O(n^3) < O(2^n)$$

**Figura 1: Gráfica general de las funciones f(n) = 0.5 - f(n) = log(n) - f(n) = n**

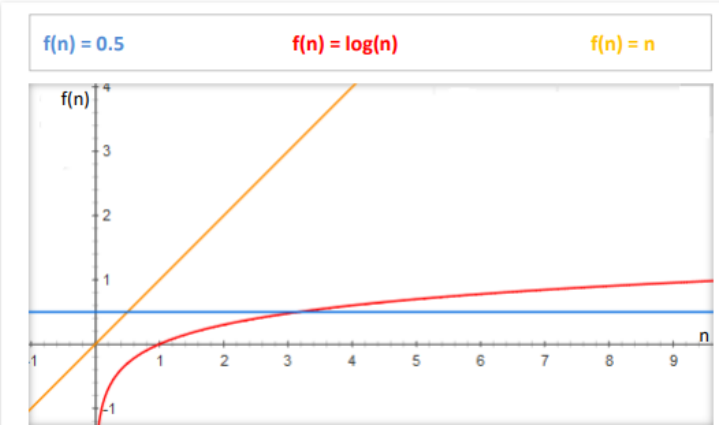
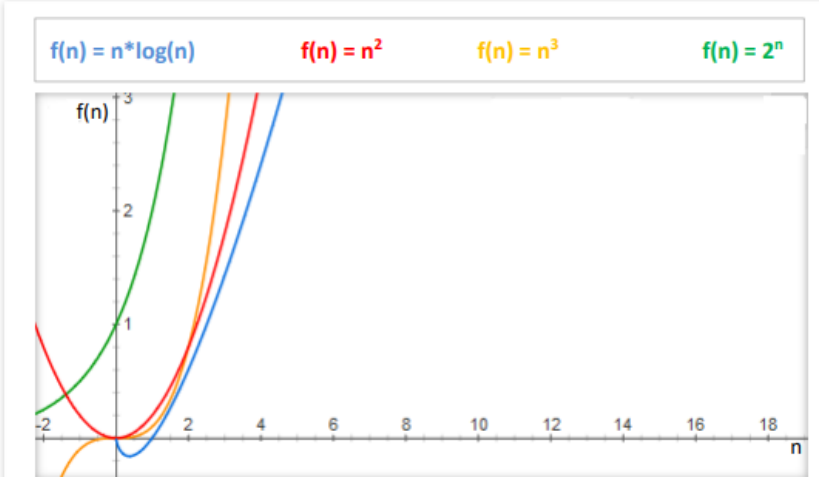



**Figura 2: Gráfica general de las funciones f(n) = n*log(n) - f(n) = n² - f(n) = n³ - f(n) = 2ⁿ**

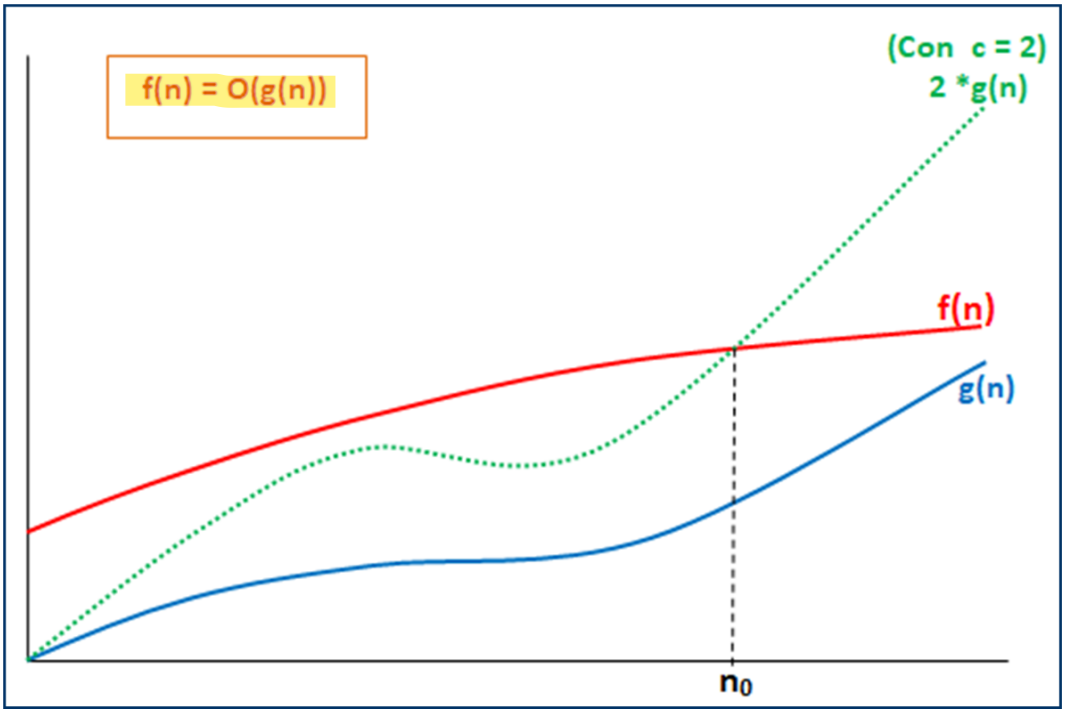



## FORMALIZACION DE LA NOTACION BIG O

La notación *Big O* se wusa para indicar un límite o cota superior para el comportamiento de una función. Si se dice que un algoritmo de ordenamiento tiene un tiempo de ejecución en el peor caso de $O(n^2)$, de alguna forma se está diciendo que ese algoritmo *no se comportará peor* que $n^2$ (o alguna función múltiplo de $n^2$) en cuanto al tiempo de ejecución.

Formalmente, decir que una función *f* **está en el orden** de otra función *g*, implica afirmar que eventualmente, a partir de cualquier valor suficientemente grande de *n*, la función *f* siempre será *menor o igual* que la función *g* multiplicada por alguna constante *c* mayor a cero. En símbolos:

Si $f(n)$ es $O(g(n)) \Rightarrow f(n) \leq c*g(n)$
(para todo valor n suficientemente grande y algún $c > 0$)




Observaciones :

1.) Toda función f polinómica en n de grado k, es orden $n^k$.

Ejemplos: Si **f(n) = ½ $n^2$ - ½ n** entonces **f(n) = O($n^2$)**

Si **f(n) = 3$n^4$ + 2$n^2$ - 5** entonces **f(n) = O($n^4$)**

**No es necesario escribir la función completa en notación Big O: elimine las constantes y quédese con el término dominante.**

2.) Toda función f exponencial en n + k, es orden $2^n$.

Ejemplo: Si **f(n) = $2^{n+10}$** entonces **f(n) = O($2^n$)**

**En notación Big O, no se complique innecesariamente con las constantes que se sumen a n en la función exponencial: elimine esas constantes y quédese con el factor dominante $2^n$.**

3.) Toda función f logarítmica en n, es orden $\log_2(n) = \log(n)$.

Ejemplo: Si **f(n) = $\log_4(n)$** entonces **f(n) = O($\log_2(n)$) = O(log(n))**

**En notación Big O, la base del logaritmo no es relevante y puede obviarse.**

CONSIDERACIONES PRACTICAS

Van ahora algunos consejos prácticos para intentar plantear una función o relación de orden para el tiempo de ejecución de un algoritmo dado...

1.Dado el algoritmo a analizar (diagrama, pseudocódigo, programa) tenga claro el factor a analizar: tiempo o memoria.

2.Determine el tamaño del problema: la variable que indica cuántos datos serán procesados.

3.Si necesita un conteo exhaustivo o riguroso, tendrá que aplicar conocimientos y pasos matemáticos detallados y cuidadosos para dar con la fórmula buscada...

4.Si le basta un análisis asintótico, y ya tiene una función de conteo exhaustiva, pásela a notación Big O: quédese con el término dominante y prescinda de constantes superfluas. En general, no diga que el tiempo es $O(n^3 + n^2)$: el término $n^2$ en el infinito es irrelevante frente a $n^3$, y por lo tanto su tiempo asintotico es $O(n^3)$.

5.Si no tiene una función de conteo exhaustiva, identifique la estructura general del algoritmo con respecto a la operación crítica, e intente hacer un conteo en forma general. Ayúdese con estos patrones:

*Operación crítica en **un sólo** ciclo de n repeticiones: $t(n) = O(n)$.

*Operación crítica dentro de **dos** ciclos anidados de **n** repeticiones cada uno: $t(n) = O(n^2)$. Y si tiene **k** ciclos anidados, entonces $t(n) = O(n^k)$.

Si el algoritmo consta de varios bloques independientes entre sí, con tiempos de ejecución diferentes, entonces el tiempo de ejecución asintótico es el que corresponde al bloque con tiempo mayor. Por ejemplo, si en su algoritmo hay un primer bloque que ejecuta en tiempo $O(n)$ y luego dos bloques más que ejecutan en tiempos $O(n^2)$ y $O(\log(n))$, entonces el tiempo completo sería $t(n) = O(n) + O(n^2) + O(\log(n))$ pero esto es asintóticamente igual a $O(n^2)$, por lo que $t(n) = O(n^2)$

*Si todo su algoritmo es un único bloque de instrucciones de *tiempo constante* (asignaciones o condiciones, por ejemplo) sin ciclos ni procesos ocultos dentro de una función, entonces todo el algoritmo ejecuta en tiempo constante $t(n) = O(1)$ (ya que sería $t(n) = O(1) + O(1) + \ldots + O(1)$ que es lo mismo que $t(n) = O(1)$.

*Si su algoritmo toma un bloque de *n* datos, lo divide en 2, aplica una operación de tiempo constante y luego sigue con sólo una de las mitades y la vuelve a dividirla en 2, y así hasta no poder hacer otra división, entonces su algoritmo tiene tiempo $t(n) = O(\log_2(n))$ que es lo mismo que $t(n) = O(\log(n))$ (la base del logaritmo se puede obviar).

No incluya constantes en la expresión de orden, salvo las del exponente del término dominante. En general no dirá $O(2n)$ sino simplemente $O(n)$. La notación O rescata la forma de variación del término dominante y por eso las constantes pueden despreciarse

OTRAS NOTACIONES.

*La notación Big O permite estimar una cota superior para el comportamiento de una función: Si f(n) es O(g(n)), entonces f(n) será siempre menor o igual que g(n) (o que algún múltiplo de g(n)) a medida que n se hace cada vez más grande.

*Pero a veces se necesita encontrar una cota inferior para una función, o se necesita acotar tanto en forma superior como en forma inferior a esa función, y para ello se tienen otras relaciones de orden, que mostramos en la siguiente tabla:

| *Expresión* | *Significado* |
|---|---|
| f(n) es O(g(n)) | El crecimiento de f(n) es ≤ que el crecimiento de g(n) |
| f(n) es Ω(g(n)) | El crecimiento de f(n) es ≥ que el crecimiento de g(n) |
| f(n) es Θ(g(n)) | El crecimiento de f(n) es = que el crecimiento de g(n) |
| f(n) es o(g(n)) | El crecimiento de f(n) es < que el crecimiento de g(n) |

## FICHA 26: ESTRATEGIAS DE RESOLUCIÓN DE PROBLEMAS: RECURSIVIDAD

Recursividad: se está haciendo referencia a una muy particular forma de expresar la definición de un objeto o un concepto. Esencialmente, una definición se dice recursiva si el objeto o concepto que está siendo definido aparece a su vez en la propia definición

ejemplo : Una frase es un conjunto de palabras que puede estar vacío, o bien puede contener una palabra seguida a su vez de otra frase

Al trabajar con con definicion recursiva debo tener cuida con los siguientes elementos:

a) Una definición recursiva correctamente planteada, exige que la definición agregue conocimiento respecto del concepto u objeto que se define. No basta con que el objeto definido aparezca en la definición, pues si sólo nos limitamos a esa regla podrían producirse definiciones obviamente verdaderas, pero sin aportar conocimiento alguno. Por ejemplo, si decimos que: Una frase es una frase.

   no cabe duda en cuanto a que esa afirmación es recursiva, pero de ninguna manera estamos definiendo lo que es una frase. En todo caso, lo que tenemos es una identidad, y no una definición. En cambio en la definición original que dimos de la idea de frase, encontramos elementos que nos permiten construir paso a paso el concepto de frase, a partir de la noción de frase vacía y la noción de palabra, y esos elementos son los que agregan conocimiento al concepto.

b)Por otra parte, una definición recursiva correctamente planteada debe evitar la recursión infinita que se produce cuando en la definición no existen elementos que permitan cerrarla lógicamente. Se cae así en una definición que, aunque agrega conocimiento, no termina nunca de referirse a sí misma. Por ejemplo, si la definición original de frase fuera planteada así: Una frase es un conjunto que consta de una palabra seguida a su vez de una frase

tenemos que esta nueva definición es recursiva y aparentemente está bien planteada, por cuanto agrega conocimiento al indicar que una frase consta de palabras y frases. Pero al no indicar que una frase puede estar vacía, la noción de frase se torna en un concepto sin fin: cada vez que decimos frase, pensamos en una palabra, y en otra frase, lo cual a su vez lleva a otra palabra y a una nueva frase, y así sucesivamente, sin solución de continuidad

Las definiciones recursivas no son muy comunes en la vida cotidiana porque en principio, siempre existe y siempre resulta más simple pensar y entender una definición directa, sin la vuelta hacia atrás que supone la recursividad. La misma noción de frase, puede definirse sin recursividad de la forma siguiente:

Una frase es una sucesión finita de palabras, que puede estar vacía

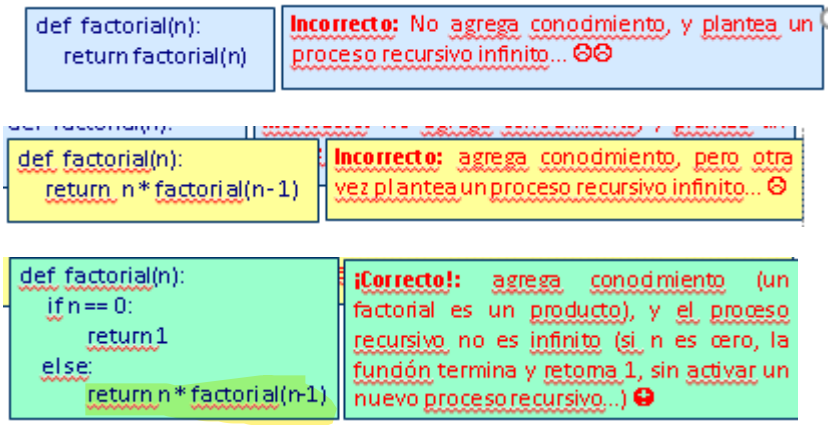
PROGRMACION RECURSIVA

. En Python, la idea es que un algoritmo recursivo puede implementarse a través de funciones de comportamiento recursivo [2]. En términos muy básicos, una función recursiva es una función que incluye en su bloque de acciones una o más invocaciones a sí misma. En otras palabras, una función recursiva es aquella que se invoca a sí misma una o más veces. En esencia, entonces, la siguiente función es recursiva (aunque aclaramos que está mal planteada):

```
def procesar():
        procesar()
```

una de las condiciones para  que sea recurisva es que se invoque a si misma, en este caso, se esta cumpliendo.pero por lo poco que vimos sabes que esta mal, . un breve análisis inmediatamente permite deducir que una vez invocada esta función provoca una cascada infinita de auto-invocaciones, sin realizar ninguna tarea útil. Como ya veremos, este proceso recursivo infinito provocará tarde o temprano una falla de ejecución por falta de memoria, y el programa se interrumpirá.

Para poner un ejemplo conocido, supongamos que se desea plantear una función recursiva para calcular el factorial de un número n que entra como parámetro. Si planteáramos una función factorial(n) para calcular recursivamente el factorial de n, pero lo hiciéramos a la manera incorrecta que vimos antes para procesar(), quedaría:




SEGUIMIENTO DE LA RECURISIVIDAD

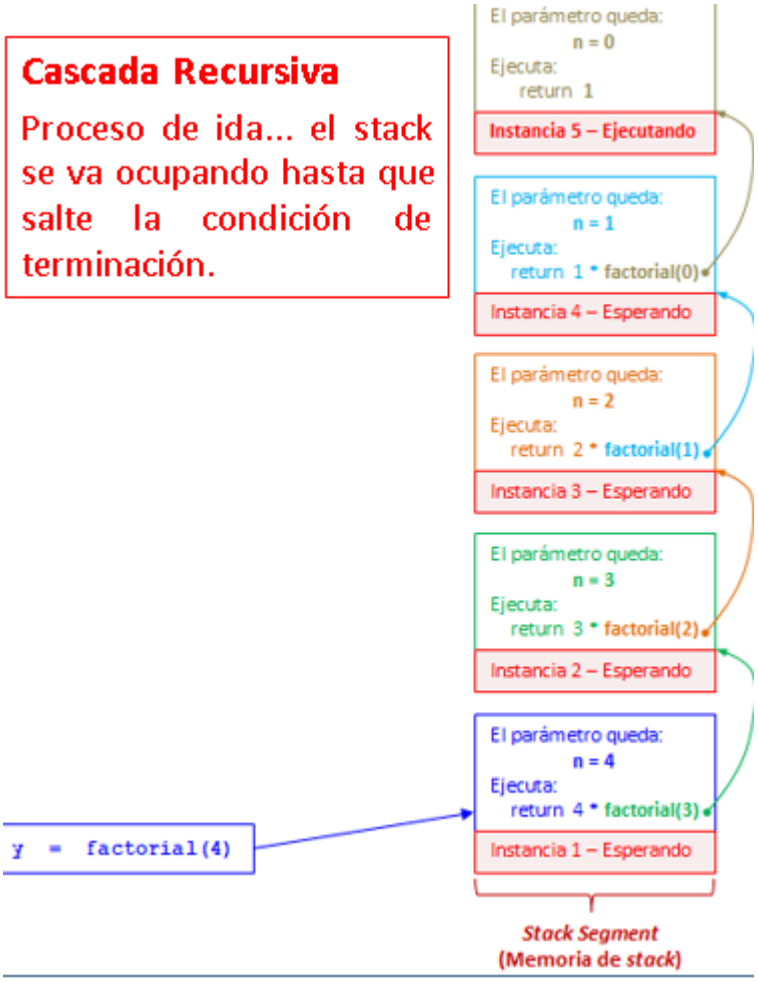
Cuando una función es invocada (recursiva o no), se le asigna un pequeño bloque de memoria en un segmento de la memoria principal conocido como ***Stack Segment*** (o ***Segmento Pila***). En ese bloque, la función almacena sus variables locales, y su dirección de retorno (la dirección de memoria a la que debe volver cuando termine).

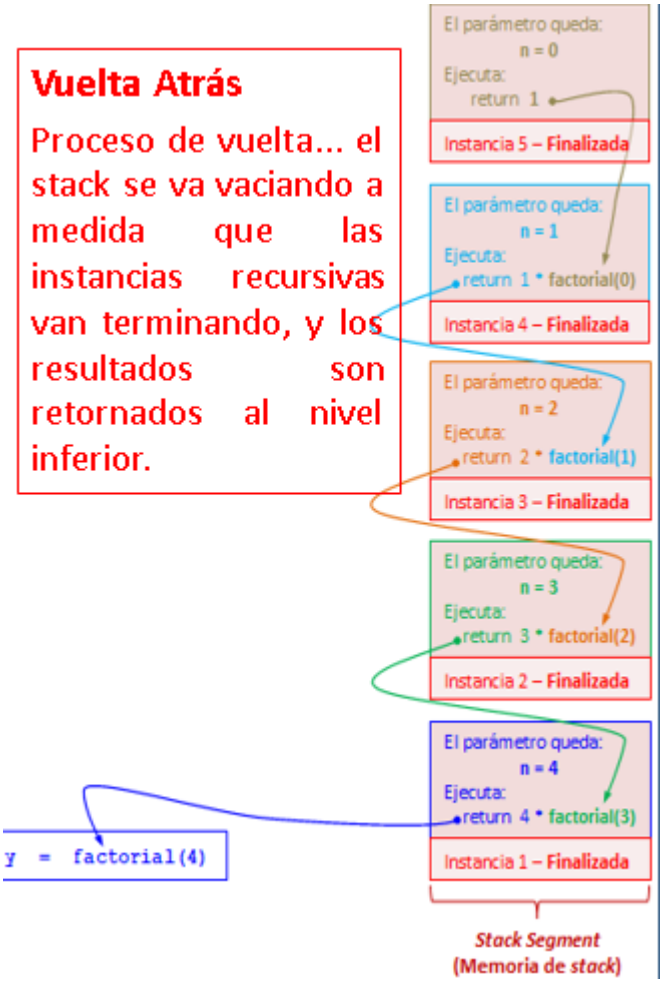
Por lo tanto, una función (recursiva o no) ocupa cierta cantidad de memoria extra cuando se activa. Si una función a su vez invoca a otra, esa otra también recibe un bloque en el ***Stack***, que se ubicará encima del bloque anterior, en modo LIFO (y de ahí el nombre de ***Stack Segment***).

La última función en invocarse estará en la cima del ***Stack***, y la función que la invocó estará inmediatamente debajo de ella en el ***Stack***. Como el ***Stack*** es un segmento de poco tamaño, una secuencia muy larga de funciones que se invoquen entre ellas podría desbordarlo (lo que se conoce como ***Stack Overflow***) haciendo que el programa se interrumpa.

## Cascada Recursiva

Proceso de ida... el stack se va ocupando hasta que salte la condición de terminación.






RECURSIVIDAD EN LA SUCESIÓN DE FIBONACCI

La Sucesión de Fibonacci es una secuencia de valores naturales en la que cada término es igual a la suma de los dos anteriores, asumiendo que el primer y el segundo término valen 1. El planteo formal es el que sigue (y note que es directamente recursivo...):

$$F(n) \quad (n \text{ entero}, n >= 0) \begin{cases} = 1 & (\text{si } n = 1 \text{ ó } n = 0) \\ = F(n-1) + F(n-2) & (\text{si } n > 1) \end{cases}$$

Se puede hacer un planteo iterativo (basado en un ciclo) y otro recursivo de sendas dunciones que calculen el valor del termino enesimo:

```
def fibonacci01(n):
    ant2 = ant1 = 1
    for i in range(2, n + 1):
        aux = ant1 + ant2
        ant2 = ant1
        ant1 = aux
    return ant1
```

```
def fibonacci02(n):
    if n <= 1:
        return 1
    return fibonacci02(n - 1) + fibonacci02(n - 2)
```

Tanto para el cálculo del factorial como para el cálculo del término enésimo de Fibonacci, tenemos dos funciones: una iterativa y otra recursiva... ¿Cuál de ambas versiones (en cada caso) será la más eficiente?

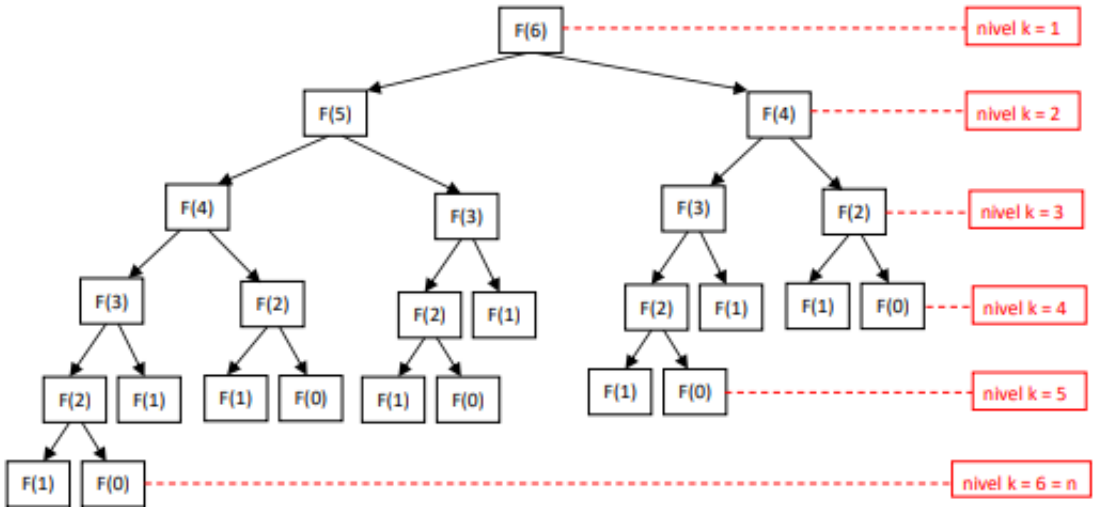
Ambas versiones del factorial ejecutan en O(n) unidades de tiempo. Pero la versión recursiva usa O(n) bloques de memoria de stack, mientras que la iterativa usa un número constante de variables para todo el cálculo, por lo que su consumo de memoria es O(1). La versión iterativa es entonces la más conveniente...

El análisis comparativo de las versiones para Fibonacci es algo más complejo... Intentaremos primero ejecutar ambas versiones en PyCharm, y luego completaremos el análisis

(esto es en python)

FIBONACCI: ITERATIVO VS RECURSIVO

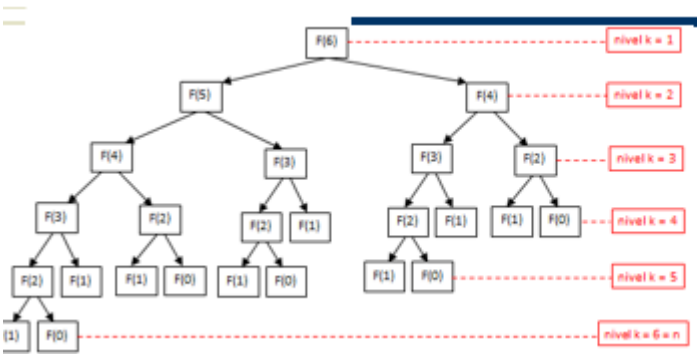
**Figura 6: Esquema del *árbol de llamadas recursivas* para *F(6)*.**




En cuanto a la memoria, la versión iterativa insume un puñado fijo de variables locales, por lo que tenemos O(1) unidades de memoria. La versión recursiva tiene que alojar un bloque en el stack por cada caja del diagrama. pero el detalle es que nunca están todas juntas: el máximo nivel de ocupación es que el corresponde a la rama más larga del árbol anterior (que es la rama extrema izquierda, con 6 cajas si n=6)... Por lo tanto, la versión recursiva requiere en un momento dado un máximo de O(n) unidades de memoria. Y hasta aquí es preferible la iterativa.

En cuanto al tiempo, la iterativa termina en O(n) unidades de tiempo, pero vimos que la recursiva es sorprendentemente lenta para valores de n tan pequeños como 35 o 40... Y ni hablar de 60 o 100... ¿Qué pasó?

wLa versión recursiva aplica dos llamadas recursivas para cada cálculo, y hay que esperar a que TODAS terminen para llegar al resultado. La cantidad de cajas a ejecutar nos da una medida del tiempo a esperar...



| Nivel | Invocaciones hasta ese nivel (incluido) |
|---|---|
| k = 1 | $1 = 2^1 - 1$ |
| k = 2 | $3 = 2^2 - 1$ |
| k = 3 | $7 = 2^3 - 1$ |
| k = 4 | $15 = 2^4 - 1$ |
| k = 5 | 23 |
| k = 6 | 25 |

La cantidad de cajas crece con el número de niveles, y la progresión esconde un crecimiento exponencial $O(2^n)$. Por lo tanto, la versión recursiva tiene un tiempo de ejecución que la hace directamente inaplicable a partir de valores pequeños de n como 35 o 45. Y para valores como 60 o más, la demora es tan asombrosa que toda la vida del universo podría no alcanzar para llegar al resultado final...

A MODO DE CONCLUSION

La recursividad es muy útil para el planteo de algoritmos, pero debe ser usada con cuidado y con conocimiento adecuado en cuanto a la forma de estimar el uso de recursos de tiempo y memoria.

wEn general, desde el punto de vista de la *complejidad del código fuente*, la recursión permite escribir programas más compactos, más simples de comprender, y más consistentes con la definición formal del problema que en un planteo iterativo. Pero si se toma a la ligera el consumo de tiempo y memoria usada, el programa obtenido podría ser inaceptable en la práctica.
Podría alegarse que entonces no es conveniente aplicar recursividad *en ningún caso*. Sin embargo, si la recursión se aplica como parte de alguna estrategia más general (como ***Divide y Vencerás*** o como ***Backtracking***), puede ayudar al planteo de soluciones muy eficientes. Y existen problemas y campos donde la recursión aplicada en forma directa también es muy eficiente (como el recorrido de estructuras de datos no lineales, o la generación de gráficos fractales, por ejemplo