

FINAL AED

Tabla de contenido

Ficha 1 FUNDAMENTOS	1
CUESTIONARIO	5
FICHA 2 Estructuras Secuenciales	15
CUESTIONARIO	18
FICHA 3 Tipos Estructurados básicos en Python	26
DESAFIO	32
CUESTIONARIO	34
Ficha 4 Estructuras Condicionales.....	41
CUESTIONARIO	49
FICHA 5 Estructuras Condicionales: Variantes	58
CUESTIONARIO	61
FICHA 11 Módulos y Paquetes	67
CUESTIONARIO	77
FICHA 12 Arreglos Unidimensionales.....	85
CUESTIONARIO	92
FICHA 13 Arreglos. Algoritmos y Técnicas Básicas	99
CUESTIONARIOS	105
FICHA 14 Arreglos: Caso de Estudio I	112
CUESTIONARIO	117
Fichas Nro. 20 Análisis de Algoritmos	128
CUESTIONARIO	134
FICHA 21 Formalización de algoritmos	140
CUESTIONARIO	144
FICHA 22 Archivos	150
CUESTIONARIO	159
FICHA 23 Archivos: Gestión ABM	166
CUESTIONARIO	171

Ficha 1 FUNDAMENTOS

Programa: está conformado por una serie de órdenes para resolver una problemática y puede adaptarse para resolver diferentes problemas

Algoritmo: es un conjunto finito de pasos que permiten resolver un determinado problema.

Estructura de un algoritmo



Los algoritmos tienen ciertas propiedades:

- Final previsible (conjunto finito de pasos)
- No es necesario volver a pensar la solución del problema una vez desarrollado el algoritmo.
- Conformado por pasos que se basan en un conjunto mínimo de acciones válidas denominadas conjunto de instrucciones primitivas (pueden ser eventualmente automatizados).

Las instrucciones primitivas son aquellas que mínimamente se esperan sean comprendidas por quien ejecuta el algoritmo.

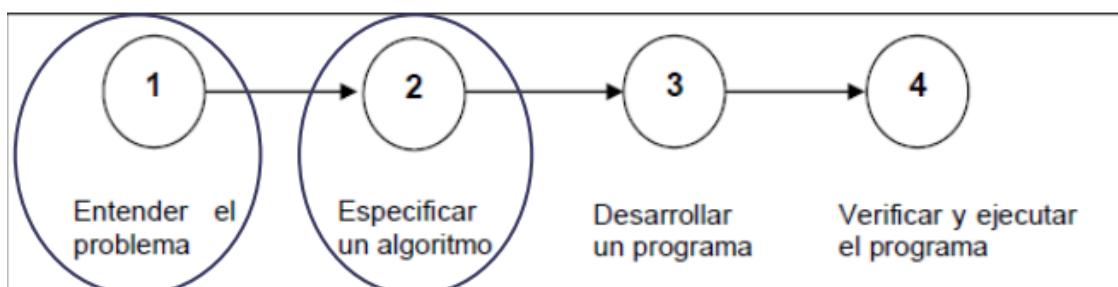


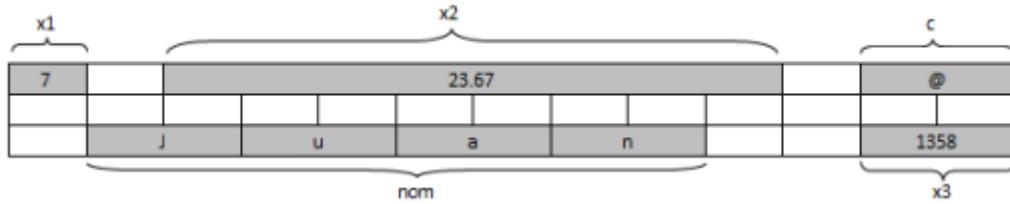
Figura 2: Pasos para resolver un problema mediante un computador

El programa que escribe un programador usando un determinado lenguaje de programación se denomina código o programa fuente.

Este programa fuente debe ser “controlado” para determinar la existencia de errores de sintaxis (errores de compilación). Para ello existen programas denominados compiladores o intérpretes.

Lenguaje de Programación

- Un lenguaje de programación esta conformado por una serie de componentes:
 - Palabras válidas (palabras reservadas)
 - Conjunto de símbolos para operaciones matemáticas, lógicas (operadores)
- Todo lenguaje de programación define su sintaxis. Esta especifica las reglas específicas de como deben combinarse operadores y las palabras reservadas.
- Cada lenguaje de programación almacena los datos dentro de la computadora mas precisamente en la memoria. Las instrucciones del programa operan con esos datos y obtienen resultados que también se alojan en memoria.
- Variable: es un grupo de bytes asociado a un nombre o identificador. Por medio de dicho nombre se puede usar o modificar su contenido. Estas se almacenan en memoria.



Para la denominación de una variable se recomienda respetar ciertas reglas:

- | El nombre de una variable en Python, solo puede contener letras del alfabeto inglés.
- | El nombre NO debe empezar con dígito.
- | El nombre de una variable puede contener cualquier cantidad de caracteres de longitud.
- | Las palabras reservadas en Python no pueden usarse como identificadores.
- | Python es case sensitive.

Elementos básicos de programación: Tipo de datos.

- **Tipo de Dato:** es la clase de valores que una variable puede contener en un momento dado.
- En la siguiente tabla se resume los tipos de datos elementales del lenguaje Python

Tipo o Clase	Descripción	Bytes por c/ variable	Rango
bool	Valores lógicos	1	[False,True]
int	Números enteros	Dinámico	ilimitado
float	Números reales	8	hasta 15 decimales
str	Cadena de caracteres	2*cantidad de caracteres	Unicode

Asignación: es un mecanismo usado para dar un valor a una determinada variable. El operador **(=)** se designa como operador de asignación. Ejemplo:

```
# se crea una variable int con un valor igual a 21.
```

```
edad = 21
```

- Python es un lenguaje de tipado dinámico. Significa que una variable puede asumir distintos tipos de datos en diferentes momentos de ejecución del programa. Ejemplo:

```
# se crea una variable int con un valor igual a 20.
```

```
n = 20
```

```
# ahora el valor de n cambia a 'Juan' (una cadena).
```

```
n = 'Juan'
```

```
# y ahora se vuelve a cambiar el valor de n a un lógico (boolean)  
n = False
```

print() cuyo propósito es la visualización de resultados.

```
print('El resultado de la suma es:', b)
```

input() operaciones de carga de datos desde el teclado mientras el programa se está ejecutando.

Lo que el usuario cargue por teclado será retornado por la función input() en forma de “cadena caracteres”.

int(cadena): retorna el número entero representado por la cadena tomado como parámetro.
float(cadena): retorna el número en coma flotante representado por la cadena tomada como parámetro

Ejemplo a): ingresar un dato de entrada => entero

```
n = int(input('Ingrese un valor entero:'))
```

Ejemplo b): ingresar un dato de entrada => flotante

```
x = float(input('Ingrese un valor en coma flotante:'))
```

Operadores Aritméticos en Python

Los operadores son símbolos que actúan sobre el valor de una variable o una constante, aplicando operaciones aritméticas básicas:

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

Tabla 2: Operadores aritméticos básicos en Python

CUESTIONARIO

para cada una de las personas célebres cuyos nombres aparecen a la izquierda, seleccione el aporte principal que dicha persona ha realizado al mundo de las ciencias informáticas o las ciencias exactas.

Thomas Flowers → Diseñador de Colossus, la primera máquina operable considerada como antecedente de las computadoras modernas.,

Ada Byron → Primeros conceptos fundamentales de programación (subrutinas, ciclos, etc.),

Alan Turing → Director del equipo que desarrolló Bombe, la máquina que permitió descifrar el código Enigma alemán.,

Abu Abdallah Muḥammad ibn Mūsā al-Jwārizmī (Abu Yāffar) → Primeras reglas algorítmicas para las operaciones aritméticas elementales en números

Charles Babbage → Diseño de la Analytical Engine (primer diseño práctico de una computadora en el mundo)

¿Cuál es el problema (si lo hay) si se ejecuta el siguiente script en Python 3?

```
n1 = 10
n2 = 14
n1 = int(input('Ingrese un número entero: '))
n2 = float(input('Ingrese un número en coma flotante: '))
print('n1: ', n1)
print('n2: ', n2)
```

Seleccione una:

- a.La variable *n2* se definió como *int* al asignarle el valor inicial 14, y luego se le asignó un valor *float* cargado por teclado: no se puede cambiar el tipo de una variable.
- b.Si una variable ya fue asignada con un valor, no se puede cambiar ese valor por otro cargado por teclado.
- c.No hay ningún problema.

¡Ok! Si tuvo alguna duda por el hecho de que la variable *n2* comenzó con un valor *int* y luego se le asignó un valor *float*, no olvide que Python es un lenguaje de tipado dinámico, y por lo tanto una variable puede cambiar de tipo durante la ejecución de un programa.

- d.En Python 3 no hay ninguna función llamada *float()* para convertir cadenas a números flotantes.

¿Cuántos números enteros diferentes pueden representarse en binario si se dispone de un conjunto de 8(ocho) bytes agrupados?

Seleccione una:

- a.2 (= 18446744073709551616 números enteros diferentes)

¡Ok! Cada bit permite 2 combinaciones, y dispone de (8 bytes * 8 bits cada uno) = 64 bits en total, por lo que la cantidad total de combinaciones (o sea, la cantidad de números diferentes que pueden representarse) es 2 .

- b.64 (= 4096 números enteros diferentes)
- c.2 (= 65536 números enteros diferentes)
- d.2 (= 256 números enteros diferentes)

Suponga la siguiente instrucción de carga por teclado en Python 3:

```
x = float(input('Ingrese un numero: '))
```

¿Cuál de las siguientes afirmaciones es **CIERTA**?

Seleccione una:

- a.Si se ingresa por teclado un número entero, la variable **x** quedará valiendo el valor *None*.
- b.Si se ingresa por teclado un valor que no puede convertirse a un número, se producirá un error y la ejecución del script se interrumpirá.
- c.Si se ingresa por teclado número entero, se producirá un error y la ejecución del script se interrumpirá.
- d.Si se ingresa por teclado un valor que no puede convertirse a un número, la variable **x** quedará valiendo *None*.

Analice el siguiente script simple, cuyo objetivo es tomar por teclado los datos básicos de un postulante a un crédito, y mostrar por consola estándar los datos cargados:

```
a      =      input('Ingrese su nombre: ')
print('El nombre ingresado es: ', a)
a      =      int(input('Ahora ingrese su edad: '))
print('La edad ingresada es: ', a)
a      =      float( input('Y ahora ingrese sueldo: '))
print('El sueldo ingresado es: ', a)
```

¿Producirá algún problema la ejecución de este script?

¿Producirá algún problema la ejecución de este script?

Seleccione una:

- a.
Sí. El script comenzará a ejecutarse, pero lanzará un error y se interrumpirá cuando intente cargar la edad del postulante en la variable a que ya contenía el nombre.
-

b.

No. El script se ejecutará sin problemas y hará lo esperado.

¡Ok!

c.

Sí. El script ejecutará sin problemas aparentes, pero mostrará en consola estándar siempre el mismo valor: el valor *None*.

d.

Sí. El script ejecutará sin problemas aparentes, pero mostrará en consola estándar siempre el mismo valor: el nombre del postulante.

¿Qué significa **definir** una variable en Python?

Seleccione una:

a. Indicar su nombre y asignarle un valor. ¡Ok!

b. Indicar su nombre.

c. Indicar su tipo, su nombre y su tamaño.

d. Indicar su tipo y su nombre o identificador.

¿Hay algún problema con el siguiente script en Python 3?

```
y = x * 2  
print('Valor final: ', y)
```

Seleccione una:

a.

Lanza un error: la variable *x* no esta definida en el momento en que se multiplica por 2.

¡Ok!

b.

No hay ningún problema.

c.

La expresión $y = x * 2$ no tiene sentido en Python.

d.

Está mal realizada la visualización del resultado: en Python 3 *print* no debe escribirse con paréntesis.

Qué diferencia principal hay entre una *calculadora manual* común y una *computadora*?

(Tómese su tiempo para pensar y discutir esta pregunta... No encontrará la respuesta directamente en la Ficha 01).

a respuesta correcta es:

Las computadoras son programables, mientras que las calculadoras no.

Suponga que tiene un script Python almacenado en un archivo llamado "script.py". Suponga también que está trabajando bajo sistema operativo Windows, y que el archivo *script.py* está guardado en la carpeta "C:\Programas". Finalmente, suponga que la variable de entorno *PATH* de Windows contiene correctamente la ruta del intérprete de Python. En estas condiciones, ¿cuál de las siguientes órdenes provocará que el script sea ejecutado desde la línea de órdenes de Windows, asumiendo que la carpeta activa es la que indica en cada caso el prompt?

Seleccione una:



a.

C:\Programas>python prueba.py



b.

C:\Programas>python script



c.

C:\Program Files>python script.py



d.

C:\Programas>python script.py

¿Qué se entiende, en general, por *error de compilación*?

Seleccione una:



a.

Es un error producido por una *operación imposible de ejecutar*, aunque sintácticamente bien escrita (por ejemplo, una división por cero), que provoca que el programa se interrumpa de forma abrupta y anormal una vez que comenzó a ejecutarse



b.

Es un error en la *sintaxis* del programa, que provoca que el programa no pueda comenzar a ejecutarse (si es compilado) o no pueda seguir ejecutándose (si es interpretado) al llegar a la línea con ese error.

¡Ok!



c.

Es un error en la *lógica* del programa, que provoca que al ejecutarse el programa arroje resultados incorrectos.



d.

Es un error en el *hardware de la computadora*, que provoca una falla grave de funcionamiento de todos los programas.

Hay algún error en el siguiente script de instrucciones en Python 3?

```
nombre = input('Nombre: ')
edad = int(input('Edad: '))
print('Datos recibidos - Nombre: ', Nombre, 'Edad: ',
      Edad)
```

Seleccione una:



a.

No hay ningún error.



b.

El error es que las variables *edad* y *nombre* se definieron en minúsculas al hacer la carga, y luego se usaron con mayúscula en la primera letra (*Edad* y *Nombre*) al hacer las visualizaciones.

¡Ok!



c.

El error es el uso de la función *int()* en la segunda carga: no existe tal función en Python 3.



d.

El error es que la función *input()* de Python 3 no puede usarse para cargar cadenas de caracteres en forma directa.

Dado un algoritmo, llamamos *instrucciones primitivas* o *acciones primitivas* a aquellos pasos mínimos del algoritmo que necesariamente debe saber aplicar quien ejecute el algoritmo (por ejemplo, para hacer una suma de dos números de varios dígitos, las operaciones primitivas mas básicas son alinear los números hacia la derecha, y sumar números de un dígito).

Suponga que se quiere plantear un algoritmo para dibujar un tablero de ajedrez (sin las fichas... SÓLO el tablero). ¿Cuál de las siguientes opciones describe **mejor** el conjunto de acciones primitivas que sería necesario aplicar?

Seleccione una:



a.

{ Dibujar cuadrados (sólo el contorno) }



b.

{ Dibujar triángulos (solo el contorno) }



c.

{ Dibujar líneas rectas horizontales, Dibujar líneas rectas verticales }



d.

{ Dibujar cuadrados, Pintar por dentro un cuadrado con un color dado }

¡Ok!

¿Cuáles son los motivos por los cuales una persona que sabe resolver un problema, querría programar y usar una computadora para resolverlo?

Seleccione una:



a.

Porque sólo programando una computadora obtendrá soluciones numéricamente precisas y sin errores ni pérdida de precisión por valores decimales.



b.

Porque al programar una computadora, tendrá la garantía de una solución correcta.



c.

No hay motivos para que lo haga: Si sabe resolver el problema, no necesita una computadora y no hay motivo para usarla.



d.

Porque al programar una computadora para resolver el problema, ganará tiempo y ahorrará esfuerzo en el futuro: la computadora puede obtener las soluciones muy rápidamente, y con precisión.

¡Ok!

¿Qué relación existe entre los conceptos de algoritmo y programa?

Seleccione una:



a.

Un programa es un algoritmo que puede ser interpretado y ejecutado por un computador.

¡Ok!



b.

Son exactamente lo mismo.



c.

Un programa es un algoritmo que sólo puede ser interpretado por una persona.



d.

Ninguna relación.

¿Hay algún inconveniente en el siguiente script elemental de Python? (Suponga que no hay otras instrucciones previas al script mostrado)

```
a=10  
print(a)  
a='sol'  
print(a)  
a=True  
print(A)
```

Seleccione una:



a.

No hay ningún problema.



b.

Producirá un error al intentar ejecutar la tercera línea: *a = 'sol'*



c.

Producirá un error al intentar ejecutar la última línea: *print(A)*

¡Ok! Exactamente... la variable **A** que se intenta mostrar en la instrucción *print()* de la última línea, no existe. Python es case sensitive: la variable que existe es *a*.



d.

Producirá un error al intentar ejecutar la quinta línea: *a = True*

Qué valor queda valiendo la variable `x`, luego de la siguiente secuencia de instrucciones en Python 3?

```
p = 2
x = 20
x = p * 8
x = int(input('Ingrese un número entero: '))
x = 17 + p
x = p + 1
```

Seleccione una:

a.

No se puede saber cuál será el valor final de `x`: depende del valor cargado por teclado en la cuarta línea.

b.

20

c.

19

d.

3

¡Ok!

¿Qué efecto produce el siguiente script en Python?

```
b = 20
B = None
y = b * 3
print("Valor de y: ", y)
print("Valor de b: ", B)
```

Seleccione una:

a.

Produce un error al intentar ejecutar la tercera línea: `y = b * 3` (no se puede usar el valor `None` como numérico)

b.

Ejecuta sin problemas: queda `b = None`, `y = None`, `B = None`.

c.

Ejecuta sin problemas: queda `b = 20`, `y = 60`, `B = None`.

¡Ok! Las variables **b** y **B** son diferentes, y no hay problema en asignar el valor None a una variable.



d.

Produce un error al intentar ejecutar la última línea (la variable *B* no está definida).

Hay algún error en la siguiente secuencia de instrucciones en Python?

```
b = None  
a = b + 1  
b = 1
```

Seleccione una:



a.

No hay error alguno.



b.

La variable *b* está definida, pero con el valor *None* cuando se ejecuta la segunda línea. La suma no puede ejecutarse y lanza un error.

¡Ok!



c.

La constante *None* no tiene ningún significado y no existe en Python. Lanza un error en la primera línea.



d.

El signo = usado en las tres instrucciones no es un operador válido en Python.

Suponga que está trabajando directamente con el editor del *shell de Python* (por ejemplo, a través del *IDLE GUI*). ¿Qué efecto producirá el siguiente script? (claración: los símbolos ">>>" conforman el prompt del IDLE, y **no deben ser escritos por el programador**... sólo escriba las instrucciones que se marcan abajo en color azul):

```
>>>var = 12  
>>>var
```

Observación: también suponemos que el programador escribirá el script *línea por línea presionando <Enter>* al final de cada una en el shell, y **NO** que hará "copy & paste" de este bloque en el shell.

Seleccione una:



a.

Mostrará el valor de la variable `var` (un 12) en la consola de salida.

Correcto. Si está trabajando directamente con el editor del shell, no es necesario usar la función `print()` para visualizar el valor de una variable...



b.

Ejecutará correctamente, pero mostrará en consola de salida el valor `None`.



c.

Provocará un error de ejecución en la segunda línea.



d.

Ejecutará correctamente, pero no mostrará nada en la consola de salida.

¿Cuál es el valor que terminará valiendo la variable `res` luego del siguiente bloque de instrucciones?

```
a = 15  
b = 4  
res = a // b
```

Respuesta:

Cuál es el valor que termina valiendo la variable `res` luego de la siguiente secuencia, en la que se usa el operador *resto o módulo* de una división?

```
a = 17  
b = 3  
res = a % b
```

Respuesta:

Qué valor queda valiendo la variable `a` luego de la siguiente secuencia de instrucciones en Python?

```
a = 5  
b = 3  
a = b
```

Respuesta:

En general, una **expresión** es una fórmula en la cual se usan *operadores* (como suma, resta, producto, compacción, etc.) sobre diversas variables y constantes (que reciben el nombre de *operandos* de la expresión). Son ejemplos válidos los siguientes: $3 * a + 2$, $b / c - 4$, $(7 - r) / (4 + a)$, $a > b$, $x + 2 \geq 10$.

¿Es correcta la siguiente definición?

"Una **expresión aritmética** es una expresión en la cual el resultado final es un número"

Seleccione una:

Verdadero

Falso

Es posible que la misma persona que diseña un algoritmo sea también quien ejecute ese algoritmo?

Seleccione una:

Verdadero

Falso

¿Puede decirse que un proceso planteado para que tenga un comienzo en un momento dado pero de tal forma de no detenerse jamás, es un algoritmo?

Seleccione una:

Verdadero

Falso

Suponga que se le pide desarrollar un programa que muestre en pantalla todos y cada uno de los números naturales (todos los enteros positivos) ¿Puede hacerse un programa así?

Seleccione una:

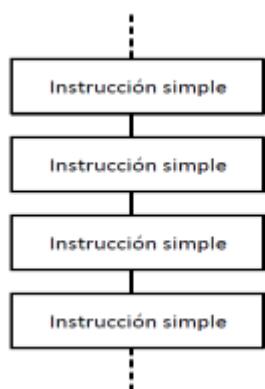
Verdadero

Falso

FICHA 2 Estructuras Secuenciales

Problema simple: no admite (o no justifica) ser dividido en problemas menores o subproblemas.

- Se caracterizan porque pueden ser resueltos mediante la aplicación de secuencias simples como:
 - Asignaciones, salidas y lecturas por consola estándar.
- Un bloque de instrucciones lineales se conoce en programación estructura secuencial de instrucciones.

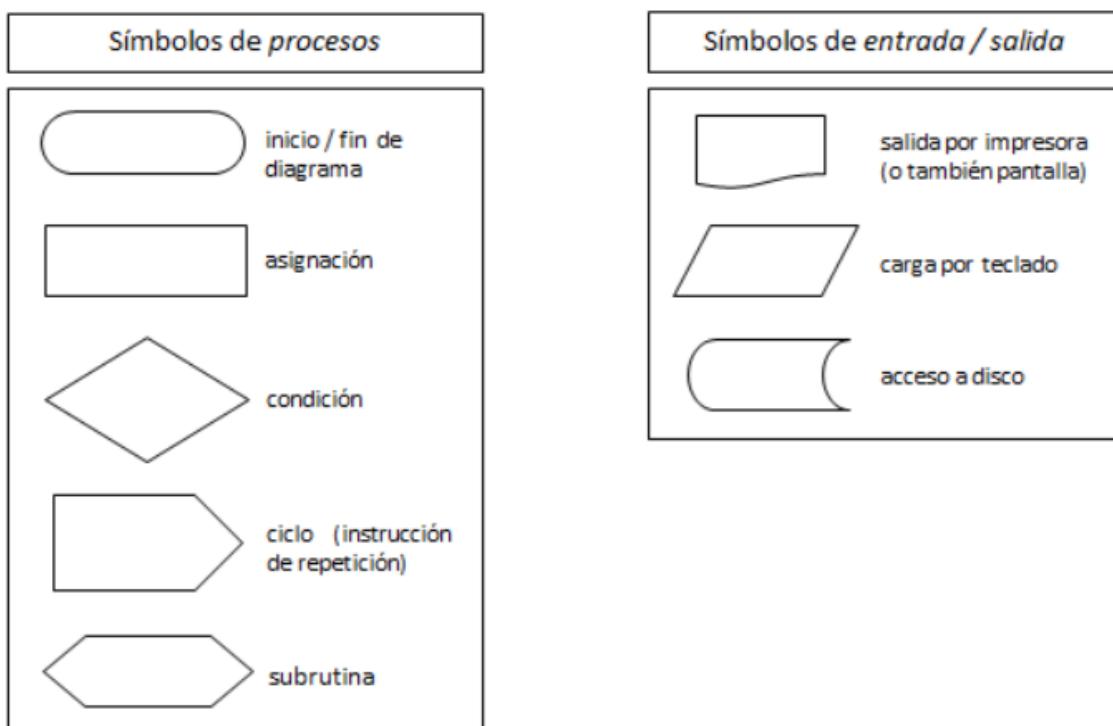


Los programadores usan lo que se conoce como un Entorno Integrado de Desarrollo (IDE).

- Un IDE incluye:
 - Un editor completo de texto.
 - Opciones de configuración de todo tipo.
 - Herramienta de compilación, depuración y ejecución.
 - Existen numerosos IDEs para Python, pero para esta asignatura usaremos el IDE PyCharm en su versión educativa.

Técnicas de representación de algoritmos: Diagrama de Flujo

- Diagrama 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 símbolos pueden clasificarse en dos grupos:
 - Símbolos de representación de proceso: representan una operación para transformar datos en resultados. Por ejemplo: el símbolo usado para representar operaciones de asignación es un rectángulo.
 - Símbolos de representación de operaciones de entrada y/o salida: representan una acción mediante la cual se lleva a cabo alguna operación de carga de datos o salida de resultados.



- Consideraciones sobre Diagrama de Flujo:

- El diagrama de flujo se construye y se lee de arriba hacia abajo.
- El diagrama de flujo comienza y termina con el símbolo de inicio/fin.
- El diagrama de flujo es genérico.
- Un único diagrama de flujo debe servir de base para un programa en cualquier lenguaje.
- No debería incluir indicaciones de visualización de mensajes en pantalla.

- Pseudocódigo: técnica que plantea el algoritmo escribiendo cada acción o paso en lenguaje natural cotidiano.

- | El pseudocódigo esta pensado para ser leído y entendido por una persona y no un computador.
 - | La palabra pseudocódigo hace referencia a una forma de código incompleto o informal.
 - | El nivel de profundidad que se refleja en el pseudocódigo depende de la necesidad del programador. Ejemplo:
- Reglas de trabajo para el uso de pseudocódigo:
 - | Indicar en la primera línea el nombre del proceso que se está describiendo.
 - | Enumerar cada paso en forma correlativa. Si hubiese subpasos, enumerarlos en base al paso principal.
 - | Mantener el encolumnado o identación tanto a nivel de pasos principales como a nivel de subpasos.
 - | Mantener consistencia.

Precedencia de operadores y uso de paréntesis en una expresión

- Expresión: es una fórmula que combina valores (constantes o variables) designados como operandos y símbolos de operaciones (suma, resta etc.)

- Variantes de expresión en Python.

```
a = int(input('A: '))
```

```
b = int(input('B: '))
```

```
# 1.) expresión asignada en una variable...
```

```
c = a + 2 * b
```

```
print('C:', c)
```

```
#2.) expresión directamente visualizada (sin asignación previa)...
```

```
print('D:', 3 * a - b / 4)
```

```
# 3.) expresión libre... el resultado se pierde Λ..
```

```
. a + b + a * b
```

En Python los operadores que tienen prioridad de ejecución son:

- ✓ multiplicación (*), división (//, /), resto (%) y potencia (**)
- ✓ y de menor prioridad los operadores de suma (+) y resta (-)

Si dos operadores tienen la misma precedencia, se aplican sobre los resultados los operadores que están más a la izquierda.

debería usar paréntesis para alterar la precedencia

- Importante: El operador `**` (potencia) tiene precedencia derecha. Si aparece en la misma expresión que otros operadores del mismo nivel de prioridad de ejecución (`*`, `/`, `//`, `%`) entonces se aplica primero `**` y luego los demás.

CUESTIONARIO

¿Cuál de las siguientes expresiones Python **NO** calcula la *raíz cuarta* del valor contenido en la variable `a`? (Suponga que `a` contiene un número positivo)

Seleccione una:



a.

`r1 = a**0.25`



b.

`r1 = pow(0.25, a)`

¡Ok! Efectivamente, la expresión está calculando el valor de **(0.25)^a** en lugar de **$a^{0.25}$** que es lo que se pedía.



c.

`r1 = pow(a, (1/4))`



d.

`r1 = a**(1/4)`

Suponga que el Departamento de Documentación del Registro Civil cuenta con 7 oficinas numeradas en forma correlativa entre 1 y 7. Cada persona que llega a realizar un trámite debe ser enviada a una de las siete oficinas y para determinar el número de la oficina se usa como dato el número de *dni* de la persona. Suponiendo que el número de dni está almacenado correctamente en la variable `dni`, ¿cuál de las siguientes expresiones calculará correctamente el número de la oficina donde debe enviarse a cada persona?

Seleccione una:



a.

`oficina = dni // 7 + 1`



b.

`oficina = dni % 7 + 1`

¡Ok! Efectivamente, el cálculo `dni % 7` entrega un valor que siempre estará en el intervalo [0, 6]... por lo tanto, si se suma 1 se ajusta el resultado para que calce en el

intervalo

[1,

7].



c.
oficina = dni % (7 + 1)



d.
oficina = dni % 7

¿Cuál de los siguientes conjuntos **NO** es la caracterización de una clase de congruencia (módulo 4)? (Recuerde que se denota como Z al conjunto de los números enteros)

Seleccione una:



a.

$$\{ 5*k + 1 \mid (\forall k \in Z) \}$$

¡Ok! Efectivamente, los números que caracterizan a este conjunto son de la forma $5*k + 1$, lo cual indica que todos ellos dejan un resto de 1 pero al dividir por 5 (y no necesariamente por 4). Por lo tanto, el conjunto mostrado es una clase de congruencia (**módulo 5**) [de hecho, Z_5] Y NO (módulo 4).



b.

$$\{ 4*k + 3 \mid (\forall k \in Z) \}$$



c.

$$\{ 4*k + 1 \mid (\forall k \in Z) \}$$



d.

$$\{ 4*k \mid (\forall k \in Z) \}$$

¿Cuál de las siguientes afirmaciones es cierta respecto de un *diagrama de flujo*?

Seleccione una:



a.

Se hace un diagrama de flujo para cada lenguaje en que se vaya a programar, aunque el problema sea siempre el mismo.



b.

Es un gráfico que permite ver claramente la lógica de un algoritmo, sin entrar en los detalles de la sintaxis de un lenguaje de programación.

¡Ok!



c.

Un diagrama es un gráfico que los profesores inventaron para torturar a los sufridos y nunca bien comprendidos alumnos.



d.

En un diagrama de flujo deben ponerse hasta los detalles mínimos: colores usados en la pantalla, mensajes aclaratorios en pantalla, comas y símbolos específicos de la sintaxis de un lenguaje, etc.

¿Cuál de las siguientes afirmaciones es **falsa** respecto de la técnica de *pseudocódigo* para representación de algoritmos?

Seleccione una:



a.

El pseudocódigo está pensado para ser leído e interpretado por una persona, y no por una computadora.



b.

El pseudocódigo puede basarse (con menor o mayor rigor) en la estructura general de un lenguaje particular, y en este caso se designa como un *pseudocódigo estructurado*.



c.

El planteo de un esquema de pseudocódigo se realiza siempre en base a reglas y estándares estrictos que los programadores deben conocer y respetar.

¡Ok! Recuerde: se pidió indicar cuál de las consignas es FALSA... y esto es efectivamente falso en cuanto al planteo de pseudocódigos.



d.

El programador es quien decide la forma, la profundidad y el nivel de detalle expresado en la lógica y en la estructura de un algoritmo.

¿Qué es en programación una *Estructura Secuencial de Instrucciones*?

Seleccione una:



a.

Una forma de organizar un conjunto de n datos para facilitar su acceso desde un programa.



b.

Una tupla compuesta por n variables del mismo tipo.



c.

Un bloque de comentarios de texto incluido en un programa.

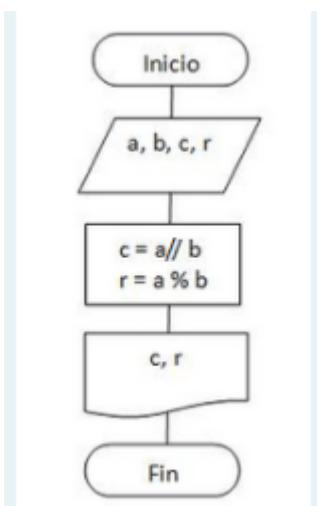


d.

Un bloque de instrucciones simples (asignaciones, visualizaciones, lecturas) escritas una debajo de la otra y ejecutadas en el orden que aparecen.

Ok.

Suponga que se desea desarrollar un programa que cargue dos números, y muestre el cociente entero y resto de la división entre esos dos números. A continuación se muestra el diagrama de flujo propuesto y el programa en Python. ¿Está bien planteado el diagrama de flujo?



```
# script en Python...
a = int(input('A: '))
b = int(input('B: '))

c = a // b
d = a % b

print('Cociente:', c)
print('Resto:', d)
```

Seleccione una:



a.

El diagrama está mal planteado (y también el programa): debió usarse una condición para verificar primero si el divisor (*b*) es menor o igual al dividendo (*a*).



b.

Sí. El diagrama está correctamente planteado.



c.

El diagrama está mal planteado: el símbolo usado al final para indicar visualización de resultados, debió ser un rectángulo.



d.

El diagrama está mal planteado: en el símbolo de carga por teclado (el paralelogramo) está indicando la *carga* de las cuatro variables, cuando solo deben cargarse dos.

¡Ok!

Suponga que las variables *a*, *b* y *c* están correctamente asignadas en forma previa, y considere la siguiente expresión en Python:

$d = a + b + c // a - b - c$

¿Cuál de las siguientes es equivalente a la expresión anterior (cuál de ellas obtiene idéntico resultado para la variable *d*)?

Seleccione una:



1.

$d = (a + b + c) // a - b - c$

Incorrecto... Revise el efecto del uso de paréntesis y la precedencia de los operadores en la expresión original... y también pruebe a desarrollar un script para probar sus conclusiones...



2.

$d = a + b + c // (a - b - c)$



3.

$d = a + b + (c // a - b - c)$



4.

`d = (a + b + c) // (a - b - c)`

La respuesta correcta es:

`d = a + b + (c // a - b - c)`

¿Hay algún inconveniente en el script Python que sigue?

```
a = int(input('A: '))
b = int(input('B: '))
c = a + b
print('Suma:', c)
```

Seleccione una:



a.

Sí. Al cargar los datos, estos se están ingresando y asignando como cadenas de caracteres, de modo que al hacer la suma el resultado será la concatenación de las cadenas en lugar de la suma de los números esperados.



b.

Sí. Está mal indentada la última línea, y provocará un error de intérprete.

Ok.



c.

Sí. Están mal usadas las funciones `input()` e `int()` (no pueden combinarse en la forma mostrada en el script).



d.

No. No hay ningún problema.

¿Qué hace el siguiente script en Python?

```
__author__ = 'Cátedra de AED'

c1 = float(input('Ingrese el primer valor: '))
c2 = float(input('Ingrese el segundo valor: '))
c3 = float(input('Ingrese el tercer valor: '))

res = (c1 + c2 + c3) / 3
print('Resultado:', res)
```

Seleccione una:



a.

Calcula y muestra el cociente entre el valor `c3` y el número `3`.



b.

Calcula y muestra el porcentaje que el valor $c1$ representa sobre el total $c1 + c2 + c3$.



c.

Calcula y muestra la suma entre los valores $c1$, $c2$ y $c3$.



d.

Calcula y muestra el promedio real de los valores $c1$, $c2$ y $c3$.

¡Ok!

¿Cuáles de las siguientes son propiedades básicas del resto de una división (y por lo tanto, aplicables al *operador resto o módulo* en un lenguaje de programación)?

Observación: note que **MAS DE UNA** respuesta puede ser correcta.
Marque **TODAS** las que considere válidas.

Seleccione una o más de una:



a.

Si se divide un número entero positivo x por otro número entero positivo n , los posibles restos son todos los números en el intervalo $[0, n-1]$ (y serían entonces, n posibles valores distintos).

¡Ok! Se deduce de la propia definición del resto de una división entre números enteros positivos.



b.

El resto de dividir un número entero positivo x por otro entero positivo n , puede ser un número mayor a n .



c.

El resto de dividir un número x por otro n , puede ser igual al número x .

¡Ok! Esto ocurrirá cada vez que $x < n$... Por ejemplo: si $x = 5$ y $n = 7$ entonces el cociente es cero... y el resto será 5...



d.

Si el resto de dividir un número x por otro n es cero, entonces x es múltiplo de n (o lo que es lo mismo, x es divisible por n).

¡Ok!

Sabemos que un *IDE* es un programa que provee herramientas para editar, depurar y ejecutar con sencillez y eficiencia un programa desarrollado en algún lenguaje de programación. El *IDE* que usaremos a lo largo del curso es el *PyCharm Edu*. Concretamente, ¿qué significa la sigla *IDE*?

Seleccione una:



a.

Integrated Database Engine (Motor Integrado de Bases de Datos)



b.

Integrated Development Environment (Entorno Integrado de Desarrollo)

Ok.



c.

Integrated Database Environment (Entorno Integrado de Bases de Datos)



d.

Integrated Development Engine (Motor Integrado de Desarrollo)

¿Cuál de los siguientes símbolos NO representa un *proceso* en un *diagrama de flujo*?

Seleccione una:

a.



b.



Incorrecto... este símbolo representa un proceso: una instrucción de invocación a una subrutina.

c.



d.



Revise la Ficha 02, página 35.

La respuesta correcta es:



¿Cuál es el valor final de la variable *res*, luego de aplicar la siguiente secuencia de instrucciones en Python? (Es recomendable que primero intente ejecutar este script y luego conteste a esta pregunta):

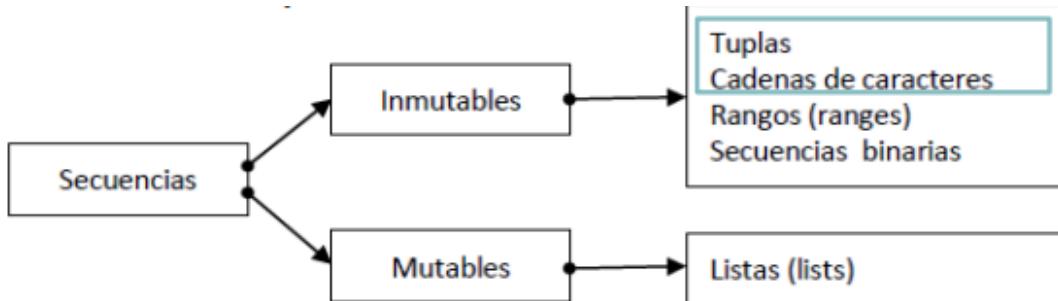
```
a = 20  
b = 6  
res = ((a // b) * 4) % 7
```

Respuesta:

FICHA 3 Tipos Estructurados básicos en Python

Tipo de dato simple: son aquellos que admiten un único valor Ejemplos: int, float y bool

Tipo de dato **compuesto**: puede contener varios valores al mismo tiempo (secuencias de datos). Ejemplos: las tuplas y las cadenas de caracteres



- **Tupla:** secuencia inmutable de datos que puede contener tipos diferentes.
- En Python hay diferentes formas de definición de una tupla “t”, que contenga ninguno, o varios valores iniciales.
- Ejemplo 1: Una tupla “t” que almacena 3 valores enteros, en este caso los ítems a almacenar se separan con comas.

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

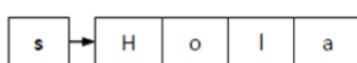


Cualquier tipo de secuencia en Python es posible utilizar la función len().

- Esta función permite obtener la cantidad de elementos de una secuencia.
- Ejemplo:

```
s = 'Hola'
m = len(s)
print(m) # muestra: 4
```

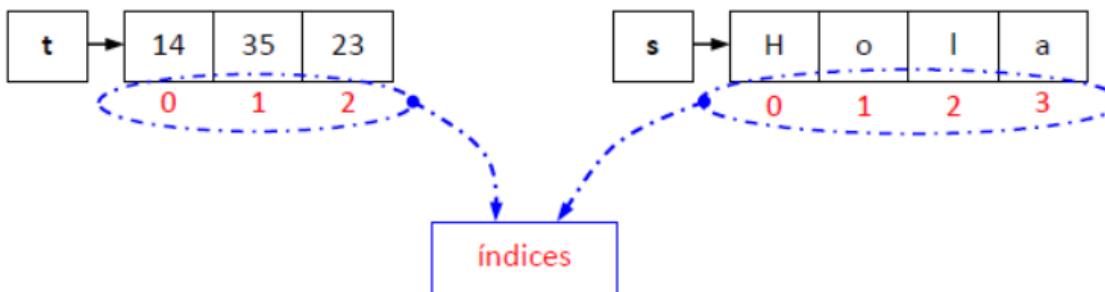
Una cadena de caracteres...



Generalidades de Secuencias en Python

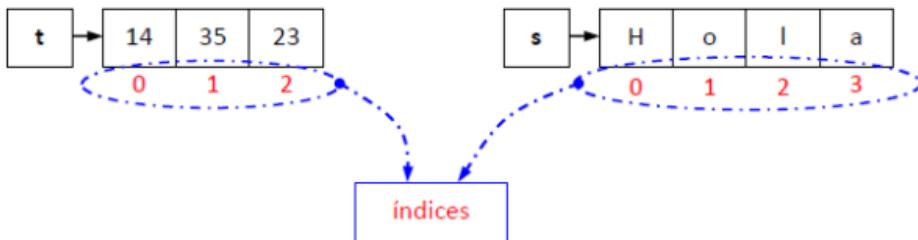
Un detalle acerca de los “tipos compuestos” o “secuencias”, es que cada casillero de una secuencia está asociado a un número llamado índice. • En Python esta secuencia de índices comienza siempre desde cero.

Dos secuencias y sus casilleros identificados por índices.



El acceso a un elemento de la secuencia es mediante el uso del índice:

Dos secuencias y sus casilleros identificados por índices.



```
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
```

Cualquier intento de modificar un casillero individual el valor de una secuencia inmutable (tupla, cadena de caracteres) producirá un error de intérprete.

```
t = 14, 35, 23
t[2] = 87  # error...
```

TypeError: 'tuple' object does not support item assignment

Un aspecto importante en el manejo de tuplas es que es válido que contenga elementos de tipos de datos diferentes. Ejemplo:

```
t2='Juan', 24, 3400.57, 'Córdoba'
print(t2)
#muestra ('Juan', 24,3400.57, 'Córdoba')
```

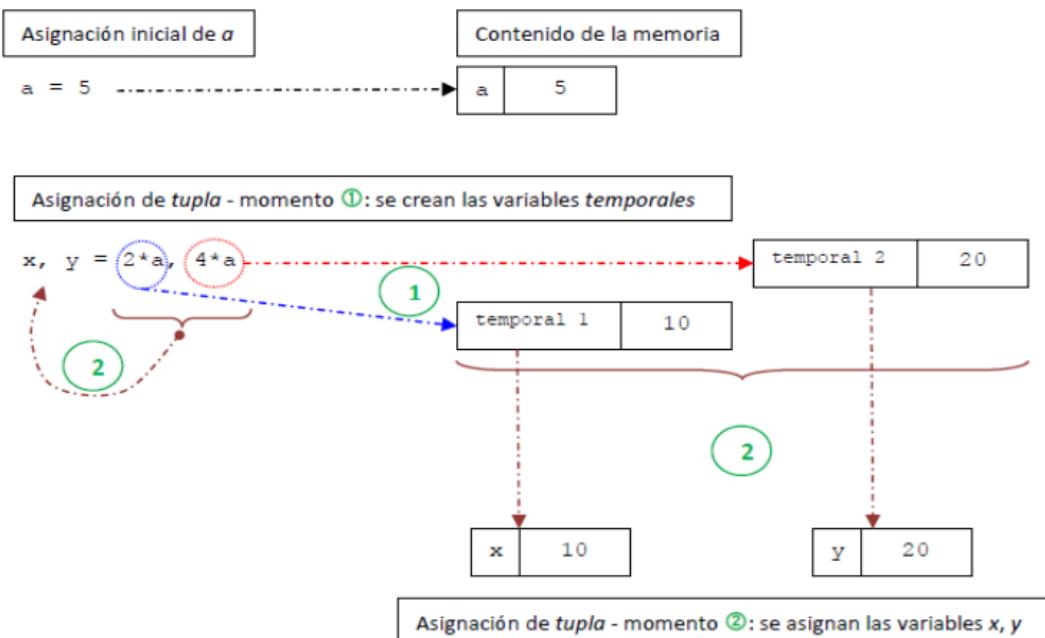
Una expresión de asignación puede estar conformada por tuplas en ambos lados de la expresión.

- Es decir una secuencia de variables (separadas por comas) del lado izquierdo del operador de asignación y otra secuencia con la misma cantidad de valores, variables y/o expresiones del lado derecho de la expresión de asignación, también separados por comas.
- Ejemplo: Las variables “x” e “y” toman respectivamente y toman respectivamente los valores 10 y 20.

`a = 5`

`x, y = 2 * a, 4 * a`

- Proceso de asignación manejo de tupla en Python.



Tuplas: Empaquetado

- El operador `(,)` actúa como empaquetador de una tupla cuando está a la derecha de la asignación. Es decir que una sola variable puede apuntar a la tupla. En este ejemplo será la variable `sec`.

```

sec = 5 , 8
print('Secuencia:', sec)
# muestra: Secuencia: (5,8)
# sec apunta a la tupla.

```

Empaquetador

Y actúa como **desempaquetador** si se encuentra a la izquierda de la expresión una tupla de variables.

```

a , b = sec
print('a:', a)
print('b:', b)
# muestra: a:5
# muestra: b:8

```

Desempaquetador

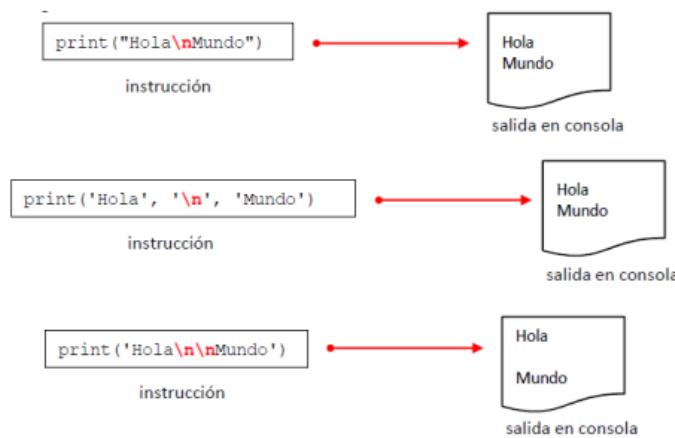
Cadenas de caracteres: Elementos adicionales

Un literal (constante) de tipo string es una secuencia o cadena de caracteres delimitada comillas dobles o por comillas simples (o apóstrofos). Ejemplos:

```
nombre = "Pedro"
```

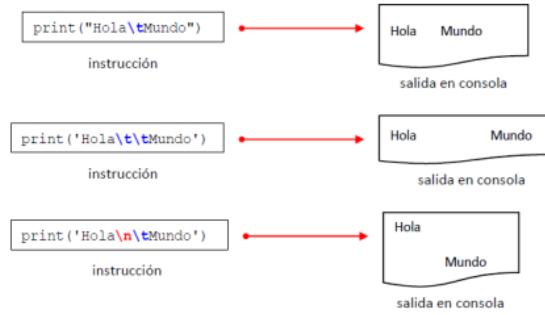
```
provincia = 'Córdoba'
```

- Python como otros lenguajes permite el uso de caracteres de escape o de control



Carácter de control \n representa salto de línea.

- Otro carácter de escape o control es `\t` provoca un espacioado en consola.

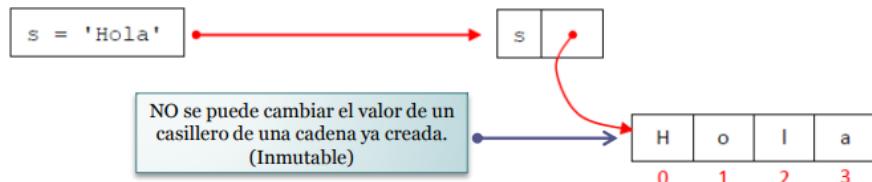


- Los caracteres de escape `\'` y `\"` representan literalmente comillas simples o comillas dobles y permiten su uso dentro de cadenas. Ejemplos:

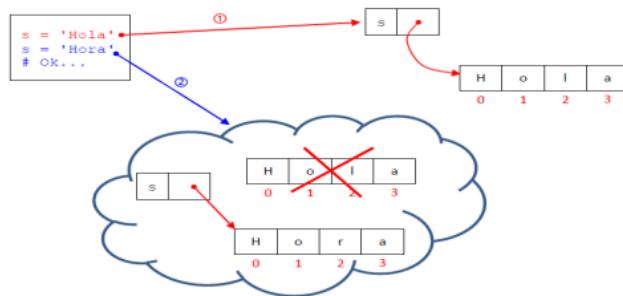
```
apellido = 'O\'Donnell' # almacena: O'Donell
autor = "Anthony \"Tony\" Hoare" # almacena: Anthony "Tony" Hoare
```

Una variable de tipo cadena (y cualquiera en gral) contiene en realidad la dirección del byte de memoria.

- Una variable de tipo cadena (y cualquiera en gral) contiene en realidad la dirección del byte de memoria.



- No existe conflicto en que una variable que apuntaba a una cadena, pase a apuntar a otra diferente.



- En Python los operadores suma (+) y producto (*) pueden usarse con cadenas de caracteres
Suma (+): permite la concatenación o unión de dos o más cadenas.

```
nombre = 'Guido'
```

```
apellido = "van Rossum"
```

```
completo = nombre + " " + apellido
```

```

print("Nombre completo: ", completo) #
muestra: Guido van Rossum

Producto(*) permite repetir una cadena varias veces concatenando el resultado.

replicado = nombre * 4

print("Nombre repetido: ", replicado)

# muestra: GuidoGuidoGuidoGuido

```

Funciones comunes de la librería estándar en Python

Función	Descripción	Ejemplo de uso
len(s)	Retorna la longitud del objeto t tomado como parámetro (un string, una lista, una tupla o un diccionario). La longitud es la cantidad de elementos que tiene el objeto t.	t = 12, 45, 73 n = len(t) print(n) # muestra: 3
max(a, b, *args)	Retorna el mayor de los parámetros tomados. Notar que admite una cantidad arbitraria de parámetros.	a, b = 6, 3 my = max(a, b) print(my) # muestra: 6
min(a,b,*args)	Retorna el menor de los parámetros tomados. Notar que admite una cantidad arbitraria de parámetros.	a, b = 6, 3 mn = min(a, b) print(mn) # muestra: 3

Función	Descripción	Ejemplo de uso
round(x,n)	Retorna el número flotante x, pero redondeado a n dígitos a la derecha del punto decimal. Si n se omite retorna la parte entera de x (como int). Si n es cero, retorna la parte entera de x (pero como float); round(3.1415) retornará 3 (int), pero round(3.1415, 0) retornará 3.0 (float).	x = 4.1485 r = round(x, 2) print(r) # muestra: 4.15
str(x)	Retorna una versión en forma de cadena de caracteres del objeto t (es decir, retorna la conversión a string de t).	t = 2, 4, 1 s = str(t) print(s) # muestra: '(2, 4, 1)'

Convención de escritura PEP-8,

Convenciones de estilo: Guía PEP-8

- Si tiene operadores con diferentes prioridades, considerar agregar espacios alrededor de los que tengan menor prioridad, y eliminar los espacios en los que tengan prioridad mayor.

Sí:

```
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)
```

- En todo caso, usar 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.

DESAFIO

#Desarrolle un programa o script Python que permita cargar por teclado un número entero que representa la cantidad de segundos que pasaron desde un evento dado.

El programa debe convertir esa cantidad de segundos a la cantidad de horas, minutos y segundos que transcurrieron. Por ejemplo, si la cantidad de segundos

ingresada es 4452 deberá

mostrar un mensaje que informe que el tiempo transcurrido fue de 1 hora, 14 minutos y 12 segundos.

#Primera ejecución - Cantidad de segundos a ingresar: 7832 a

#Segunda ejecución - Cantidad de segundos a ingresar: 18993 b

#Tercera ejecución - Cantidad de segundos a ingresar: 2475 c

#Cuarta ejecución - Cantidad de segundos a ingresar: 25213 d

```
horas = 0
```

```
minutos = 0
```

```
segundos = 0
```

```
a = int(input("» ingrese la cantidad de segundos a convertir " ))
```

#El paso a paso para convertir segundos a horas, minutos y segundos es el siguiente:

#1 Dividir los segundos entre 60. El cociente de la división serán los minutos y el resto la cantidad de segundos.

#2 Si el cociente que hemos obtenido en la división anterior era igual o mayor de 60, volvemos a dividir entre 60.

#En esta ocasión, el cociente obtenido será la cantidad de horas y el resto la cantidad de minutos.

#3 Agrupamos las horas y minutos del paso 2 y cogemos la cantidad de segundos del paso 1. Ahora ya tenemos los segundos pasados a horas, minutos y segundos.

```
print("\npasaje de ", a,"segundos")
```

```
minutos = a // 60
```

```
segundos = a % 60
```

```
if minutos >= 60:
```

```
    horas = minutos // 60
```

```
    minutos = minutos % 60
```

```
    print("horas: ", horas," minutos: ", minutos," segundos: ",segundos)
```

```
else:
```

```
    print("minutos: ",minutos," segundos: ", segundos)
```

#Primera ejecución - Cantidad de segundos a ingresar: 6551

#Segunda ejecución - Cantidad de segundos a ingresar: 21720

#Tercera ejecución - Cantidad de segundos a ingresar: 3123

#Cuarta ejecución - Cantidad de segundos a ingresar: 57862

#0:23:0

#1:14:12

CUESTIONARIO

¿Qué hace el siguiente script en Python?

```
cad = 'Python'  
print(cad[1], cad[4])  
print(cad[1] + cad[4])
```

Seleccione una:



a.

Accede al carácter en la posición 1 ('y') y al carácter en la posición 4 ('o') de la cadena. El primer print() los muestra *por separado* (y o) pero el segundo los muestra *concatenados* (yo).

¡Ok!



b.

Produce un error de intérprete, ya que no se puede acceder a los elementos individuales de una cadena mediante índices encerrados entre corchetes.



c.

Accede al carácter en la posición 1 ('y') y al carácter en la posición 4 ('o') de la cadena. Ambos print() los muestran *por separado* (y o).



d.

Accede al primer carácter de la cadena ('P') y al cuarto carácter de la cadena ('h'). El primer print() los muestra *por separado* (P h) pero el segundo los muestra *concatenados* (Ph).

¿Cuál de las siguientes asignaciones de cadenas de caracteres en una variable provocará un error de intérprete en Python?

Seleccione una:



a.

```
cad = "Ray 'Sugar' Leonard"
```



b.

```
cad = 'Ray "Sugar" Leonard'
```



c.

```
cad = "Ray \'Sugar\' Leonard"
```

d.

```
cad = \'Ray "Sugar" Leonard\'
```

¡Ok! Efectivamente... el uso del carácter de escape \ permite incluir una comilla simple dentro de una cadena, pero suponiendo que la misma ya esté delimitada con comillas o apóstrofos...

¿Hay algún problema en Python con la siguiente secuencia de instrucciones de asignación de cadenas de caracteres?

```
res = 'toro'  
print('Valor: ', res)  
  
res[0] = 'c'  
print('Valor: ', res)
```

Seleccione una:

a.

El script mostrado no lanza ningún error, pero la variable *res* termina valiendo la cadena 'toro' inicialmente asignada. La asignación del carácter 'c' será ignorada, debido a que las cadenas de caracteres son inmutables.

b.

El script mostrado lanza un error de intérprete en la asignación *res = 'toro'*: las cadenas de caracteres no pueden ser asignadas con el operador =.

c.

No. No hay ningún problema.

d.

El script mostrado lanza un error de ejecución en la asignación *res[0] = 'c'*: las cadenas de caracteres son inmutables, y por lo tanto no se puede cambiar un carácter mediante índices entre corchetes

¡Ok! Efectivamente...

Considere la siguiente instrucción de visualización en pantalla para Python 3:

```
print('Hola\nMundo...\n\t...otra vez')
```

¿Cuál de las siguientes salidas por consola estándar será la que se produzca con la instrucción anterior?

Seleccione una:



a.

Hola Mundo...
...otra vez



b.

Hola Mundo... ...otra vez



c.

Hola
Mundo...
...otra vez

¡Ok! Efectivamente... cada \n produce un salto de línea (por eso habrá tres líneas de salida) y el carácter \t produce un espaciado horizontal en la tercera antes de la cadena '...otra vez'.



d.

Hola Mundo...
...otra vez

Cuál de los siguientes **NO ES** una recomendación de estilo de escritura de código fuente en Python de acuerdo a la *Guía PEP 8*?

Seleccione una:



a.

No colocar espacios alrededor de los operadores y los operandos de una expresión: escribir la expresión de corrido, sin espacios separadores.

¡Ok! Justamente, la Guía PEP 8 sugiere lo contrario: use espaciado y aplique sentido común.



b.

Colocar los comentarios de texto en líneas específicas para esos comentarios, de ser posible.



c.

Usar líneas en blanco para separar largos bloques de código fuente.



d.

Mantener consistencia en el uso de comillas dobles o simples para manejar cadenas: si se comenzó con una de ellas, mantenerse con ella a lo largo del programa.

¿Cuál es el nombre del creador del lenguaje de programación *Python*?

La respuesta correcta es:



Guido van Rossum

¿Cuál de los siguientes es el nombre informal alternativo con el cual se conoce al documento *PEP 20* de la documentación oficial de Python? (Claramente, esta es una pregunta para distenderse... nadie espera que se equivoquen aquí... sólo tómense un minuto para buscar la respuesta... y mientras, ríanse un poco con las opciones que les estamos sugiriendo...)

Seleccione una:



a.

El Ninja Python



b.

El Python Saiyajin



c.

El Zen de Python

¡Ok!



d.

El Kung Fu Python

Qué efecto provoca la siguiente secuencia de instrucciones en Python?

```
a = 5  
b = 3  
c = a  
a = b  
b = c
```

Seleccione una:



a.

Las tres variables (a, b, c) quedan valiendo 5.



b.

Las tres variables (a, b, c) quedan valiendo 3.



c.

Quedan con los siguiente valores: **a = 5 b = 3 c = 5**



d.

Quedan con los siguiente valores: **a = 3 b = 5 c = 5.**

¡Ok!

Sabemos que **Ada Augusta Byron King (Condesa de Lovelace)** fue la hija del poeta Lord Byron y actuó como colaboradora de Charles Babbage en el diseño de la Analytical Engine, proponiendo elementos conceptuales para la programación de esa máquina que hoy se usan en los modernos lenguajes de programación (subrutinas y ciclos, por ejemplo). ¿Cómo se llama el lenguaje de programación moderno designado así en honor a ella?

Seleccione una:



a.

Byron



b.

Ada

¡Ok!



c.

Lovelace



d.

Augusta

El **Premio Turing** (otorgado anualmente por la ACM - Association for Computing Machinery a quienes hayan realizado aportes trascendentales en el campo de las Ciencias de la Computación) está considerado como el equivalente al Premio Nobel de las Ciencias de la Computación. ¿Cuál de los siguientes famosos exponentes del mundo de la Ciencias de la Computación ganó oportunamente el Premio Turing por haber sido el creador de varios lenguajes de programación innovadores (entre ellos: *Pascal, Modula y Algol-W*)?

La respuesta correcta es:



Niklaus Wirth

Hay algún problema en Python con la siguiente secuencia de instrucciones de asignación de cadenas de caracteres?

```
res = 'toro'  
print('Valor: ', res)  
  
res = 'coro'  
print('Valor: ', res)
```

Seleccione una:



a.

No. No hay ningún problema.

¡Ok! Efectivamente... la variable res comienza apuntando a la cadena 'toro', y luego pasa a apuntar a la cadena 'coro'. No hay ningún problema.



b.

El script mostrado lanza un error de intérprete en la asignación `res = 'toro'`: las cadenas de caracteres no pueden ser asignadas con el operador `=`.



c.

El script mostrado lanza un error de intérprete en la asignación `res = 'coro'`: las cadenas de caracteres son inmutables, y por lo tanto `res` no puede cambiar de valor.



d.

El script mostrado no lanza ningún error, pero la variable `res` termina valiendo la cadena 'toro' inicialmente asignada. La asignación de la cadena 'coro' será ignorada, debido a que las cadenas de caracteres son inmutables.

Cuál de los siguientes tipos estructurados es un tipo de secuencia *mutable* en Python?

Seleccione una:



a.

Tuplas (tuples)



b.

Cadenas de caracteres (str)



c.

Lista (list)

¡Ok!



d.

Rango (range)

¿Qué hace el siguiente script en Python?

```
a = int(input('a: '))
b = int(input('b: '))
c = int(input('c: '))
d = int(input('d: '))

m = min(max(a, b), max(c, d))
print('Resultado: ', m)
```

Seleccione una:



a.

Calcula la **suma** entre los valores *a* y *b*, luego la **suma** entre los valores *c* y *d*, finalmente obtiene el **menor** entre ambas **sumas** parciales, y lo muestra.



b.

Calcula el **menor** entre los valores *a* y *b*, luego el **menor** entre los valores *c* y *d*, finalmente obtiene el **mayor** entre esos dos **menores**, y lo muestra.



c.

Calcula el **menor** entre los valores *a* y *b*, luego el **mayor** entre los valores *c* y *d*, finalmente obtiene el **menor** entre los dos que obtuvo, y lo muestra.



d.

Calcula el **mayor** entre los valores *a* y *b*, luego el **mayor** entre los valores *c* y *d*, finalmente obtiene el **menor** entre esos dos **mayores**, y lo muestra.

Considere la secuencia de instrucciones Python que se muestra a continuación:

```
a = 10  
b = 20  
c = 30
```

¿Cuáles de las siguientes instrucciones o secuencias de instrucciones **NO PROVOCAN** un cambio en el valor final de la variable **b**? (Observación: más de una respuesta puede ser correcta. Marque **todas** las que considere aplicables)

Seleccione una o más de una:

a.

$d = b, a, c$
 $a, b, c = d$

b.

$d = c, a, b$
 $a, b, c = d$

c.

$a, b, c = c, b, a$

Correcto. En este caso, la única variable que no cambia de valor es justamente **b**...

d.

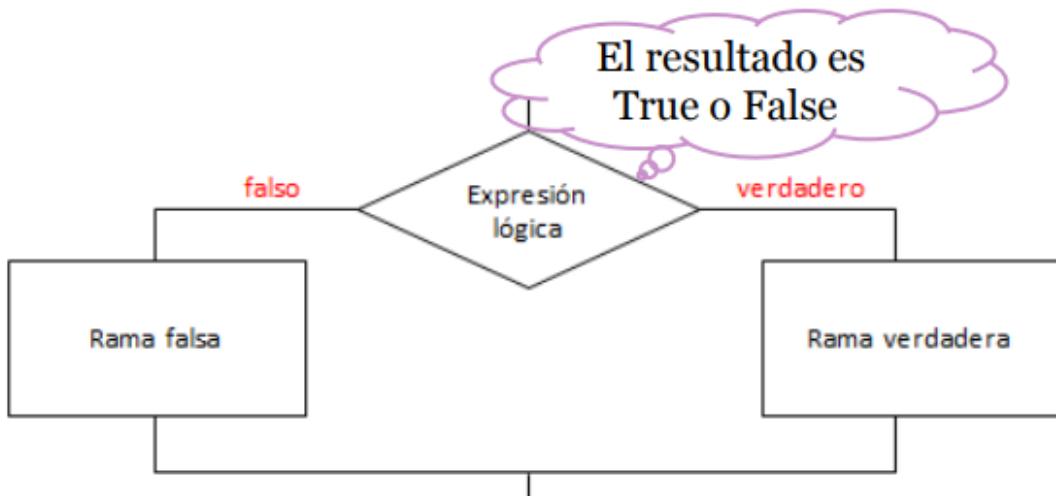
$a, b = a, b$

Ok. Tanto **a** como **b** quedan con sus valores iniciales.

Ficha 4 Estructuras Condicionales

Estructuras Condicionales: Fundamentos

- Para resolver un problema es posible que necesite comprobar el valor de alguna condición, y en función de ello realizar ciertas acciones si la condición es verdadera y otras acciones si es falsa.



Python,

if expresión lógica:

 instrucciones de la rama verdadera

else:

 instrucciones de la rama falsa

- La secuencia de instrucciones que conforman la rama verdadera y falsa se escribe respetando la indentación o encolumnado

```
a = int(input('A: '))
b = int(input('B: '))
```

```
if a > b:
    may = a
else:
    may = b
```

Nunca se ejecutan ambas ramas de una instrucción condicional !!!

```
print('El mayor es:', may)
```

En Python si usamos una estructura condicional se debe escribir cada instrucción con la indentación o encolumnado correcto

Operadores relacionales

- **Expresión:** fórmula compuesta por variables, constantes (operandos) y por símbolos de aplicación de una acción (operadores).

- Una expresión puede ser:

- **Aritmética:** el resultado final es valor numérico.

- **Lógica:** el resultado final es un valor de verdad (True o False).

- Todo lenguaje de programación provee operadores que permiten obtener un valor de verdad como resultado. Estos son llamados operadores relationales u operadores de comparación.

operadores relationales son:

Operador	Significado	Ejemplo	Observaciones
<code>==</code>	igual que	<code>a == b</code>	retorna <i>True</i> si <i>a</i> es igual que <i>b</i> , o <i>False</i> en caso contrario
<code>!=</code>	distinto de	<code>a != b</code>	retorna <i>True</i> si <i>a</i> es distinto de <i>b</i> , o <i>False</i> en caso contrario
<code>></code>	mayor que	<code>a > b</code>	retorna <i>True</i> si <i>a</i> es mayor que <i>b</i> , o <i>False</i> en caso contrario
<code><</code>	menor que	<code>a < b</code>	retorna <i>True</i> si <i>a</i> es menor que <i>b</i> , o <i>False</i> en caso contrario
<code>>=</code>	mayor o igual que	<code>a >= b</code>	retorna <i>True</i> si <i>a</i> es mayor o igual que <i>b</i> , o <i>False</i> en caso contrario
<code><=</code>	menor o igual que	<code>a <= b</code>	retorna <i>True</i> si <i>a</i> es menor o igual que <i>b</i> , o <i>False</i> en caso contrario

- En Python los operadores relationales también permite comparar en forma directa dos cadenas de caracteres.

- Los operadores `==` (igual que) y `!=` (distinto que) determina si dos cadenas son iguales o distintas.

```

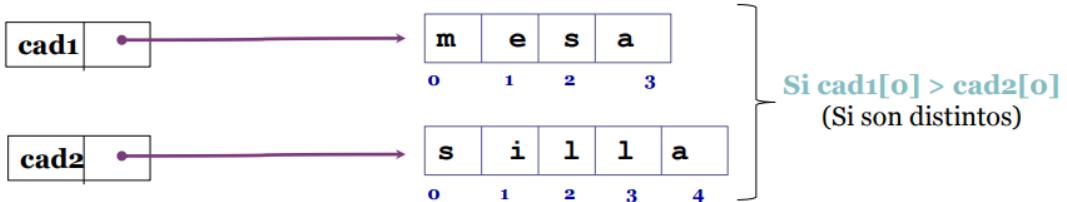
cad1 = 'Hola'
cad2 = 'Mundo'
if cad1 == cad2:
    print('Son iguales')
else:
    print('No son iguales')

if cad1 != 'Hello':
    print('No es la palabra Hello...')
else:
    print('Es la palabra Hello...')
```

- En el caso de usar los operadores `<,<=,>, o >=` para comparar cadenas, Python hará la **comparación lexicográfica**.

- Script que realiza la comparación lexicográfica entre dos cadenas:

```
cad1 = 'mesa'
cad2 = 'silla'
if cad1 < cad2:
    print('Orden alfabético:', cad1, cad2)
else:
    print('Orden alfabético:', cad2, cad1)
```



Python considera menor a la cadena que comience con el carácter que aparezca primero en la tabla Unicode. (En este caso 'm')

- Un **conector lógico u operador booleano** permite encadenar la comprobación de dos o más expresiones lógicas y obtener un único resultado.

Operador	Significado	Ejemplo	Observaciones
and	conjunción lógica (y)	a == b and y != x	ver "Tablas de Verdad" en esta misma sección
or	disyunción lógica (o)	n == 1 or n == 2	ver "Tablas de Verdad" en esta misma sección
not	negación lógica (no)	not x > 7	ver "Tablas de Verdad" en esta misma sección

- En general cada una de las expresiones lógicas encadenadas por un conector lógico se denomina proposición lógica.
- Si se considera dos proposiciones lógicas cualquiera denominadas p y q, se puede visualizar mediante tablas la forma que operan el conector and.

Tabla de verdad del conector and		
p	q	p and q
True	True	True
True	False	False
False	True	False
False	False	False

- El operador and es útil cuando se necesitar estar seguro de que todas las condiciones impuestas sean ciertas.

```

if sueldo > 15000 and antiguedad >= 10:
    print('Crédito concedido')
else:
    print('Crédito rechazado')

```

- La tabla de verdad del operador or es:

Tabla de verdad del conector or		
p	q	p or q
True	True	True
True	False	True
False	True	True
False	False	False

- Un ejemplo de uso de or puede ser cuando se necesita determinar si una variable que se acaba de ingresar por teclado asume uno de varios valores posibles que se consideran correctos.

```

if opcion == 1 or opcion == 3 or opcion == 5:
    print('Opción correcta')
else:
    print('Opción incorrecta')

```

- En Python los operadores relacionales o de comparación se pueden aplicar en forma idéntica como se hace en la notación algebraica normal como:

if a > b > c:

- La misma instrucción anterior podría haber sido escrita así:

if a > b and b > c:

- Los conectores and y or aplican sobre dos o mas proposiciones lógicas. •

Si un operador aplica sobre dos proposiciones se denomina operador binario.

- El operador not (negación lógica) aplica sobre una sola proposición lógica se suele denominar operador unario.

- Una variable del tipo “boolean” para comprobar su valor en una instrucción condicional no necesita hacer explícitamente la comparación mediante operadores relacionales

```

r1 = a > b and (c == 3*d or not e != a)
if r1 == True:
    print('Verdadero...')
else:
    print('Falso...')

```

Planteo de la condición de forma explícita.
😊😊

- Este script podría haber sido escrito de la siguiente forma:

```

r1 = a > b and (c == 3*d or not e != a)
if r1:
    print('Verdadero...')
else:
    print('Falso...')

```

Replantea la condición de forma implícita.
😊😊

- Si en una expresión lógica se aplican operadores relaciones y conectores lógicos entregará un resultado True o False.
- Además el uso de conectores lógicos and, or y not convierte la expresión analizada en expresión lógica, incluso si su valor inicial no es de tipo boolean.

- Python asume que los siguientes valores son falsos:

- False
- None
- El valor numérico 0 (cero)
- Las cadenas de caracteres vacía.

Python interpreta el valor 10 como True (como es distinto de cero)

```

a = 10
b = not a
print('b:', b)

```

Resultado de la ejecución: b: False

- El uso de conectores lógicos and, or y not convierte la expresión analizada en expresión lógica, sean cuales sean los tipos de las variables sobre las que se aplican esos conectores.

Precedencia de ejecución de los operadores relacionales y de los conectores lógicos

- Se puede establecer la siguiente relación:

precedencia(connectores) < precedencia(relacionales) < precedencia(matemáticos)

- El uso de paréntesis permite al programador cambiar estas precedencias según las necesidades.
- En cualquier expresión lógica el lenguaje agrupará los términos y resolverá primero las operaciones aritméticas. Luego los operadores de comparación y finalmente los conectores lógicos.
- En cuanto a los conectores lógicos, el not es de mayor precedencia que el and, y este a su vez es de mayor que el or.

precedencia(or) < precedencia(and) < precedencia(not)

- El conector and como el or en Python actúan en forma cortocircuitada. Significa que en función del valor de la primera proposición evaluada, la segunda o restantes podrían no llegar ser evaluadas.

Conejor	Ejemplo	Efecto del cortocircuito
and	if a > b and a > c:	Si la primera proposición es <i>False</i> ($a > b$ en este caso) la segunda ($a > c$) no se chequea y la salida también es <i>False</i> .
or	if n < 0 or n > 9:	Si la primera proposición es <i>True</i> ($n < 0$ en este caso), la segunda ($n > 9$) no se chequea y la salida también es <i>True</i> .

```
# Considere estas variables con estos valores:  
a, b, c, d, e = 3, 5, 7, 2, 3  
  
# Ejemplo: valor final asignado en r1: True  
r1 = a > b and c == 3*d or not e != a  
  
# Desarrollo paso a paso: reemplazo de variables por sus valores  
# r1 = 3 > 5 and 7 == 3*2 or not 3 != 3  
  
# Desarrollo paso a paso: primero los operadores aritméticos  
# r1 = 3 > 5 and 7 == 3*2 or not 3 != 3  
# r1 = 3 > 5 and 7 == 6 or not 3 != 3  
  
# Desarrollo paso a paso: segundo los operadores relacionales  
# r1 = 3 > 5 and 7 == 6 or not 3 != 3  
# r1 = False and False or not False  
  
# Desarrollo paso a paso: tercero los not, and y or en ese orden...  
# r1 = False and False or not False  
# r1 = False and False or True  
# r1 = False or True  
# r1 = True
```

Generación de valores aleatorios

- En Python existe lo que se conoce como módulo (librería) llamado random que contiene las definiciones y funciones para poder acceder a la gestión de números pseudo-aleatorios.
- Para ello debe usarse la instrucción import random. Y para acceder a las funciones contenidas en el módulo se antepone el prefijo random. Ejemplo: script que obtiene un valor aleatorio entre 0 y 1.

```
__author__ = 'Cátedra de AED'  
import random  
x = random.random()  
print(x)
```

Retorna un número en coma flotante dentro del intervalo [0,1]

Obtiene un valor aleatorio entre 0 y 1. (Incluye como salida el 0 pero no el 1)

- En el caso de necesitar generar un valor aleatorio de tipo entero en un rango o intervalo, se puede usar la función random.randrange(a,b). Esta permite obtener un número entero pseudo-aleatorio, pero comprendido en el intervalo [a,b-1].

```
__author__ = 'Cátedra de AED'
import random
# número aleatorio entero y tal que 2 <= y < 10
y = random.randrange(2, 10)
print(y)
```

- Otra alternativa es la función random.randint(a,b) que permite hacer lo mismo que random.randrange(a,b) pero de forma que retorna un entero que estará en el intervalo [a,b] (incluirá b como posible salida).
- La función random.choice(sec) acepta como parámetro una secuencia (aquí se llama sec) (o sea una tupla, cadena, lista) y retorna un elemento cualquiera de esa secuencia.

```
__author__ = 'Cátedra de AED'

import random

sec1 = 2, 10, 7, 9, 3, 4
r1 = random.choice(sec1)
print(r1)
#Una posible salida puede ser: 10.

sec2 = 'ABCDEFGHI'
r2 = random.choice(sec2)
print(r2)
#Una posible salida puede ser: A
```

- Ejemplo: Programa que ordena dos números (distintos), pero los dos números de entrada sean generados en forma aleatoria (tomando números entre 1 y 10), en lugar de ser cargados por teclado.

```
__author__ = 'Cátedra de AED'
import random
print('los numeros de entrada son generados
aleatoriamente...')
n1 = random.randint(1,10)
n2 = random.randint(1,10)
if n1 > n2:
    may = n1
    men = n2
else:
    may = n2
    men = n1
print('Números ordenados:', men, ' ', may)
```

CUESTIONARIO

¿Cuáles fueron los aportes que realizaron *George Boole* y *Augustus De Morgan* en el campo matemático del tratamiento de las relaciones lógicas?

Seleccione una:



a.

Boole sentó las bases del Álgebra de Boole, y *De Morgan* demostró que el Álgebra de Boole es válida.



b.

Boole sentó las bases de la aritmética en sistema binario, y *De Morgan* usó el sistema binario para diseñar la primera computadora digital de la historia.



c.

De Morgan sentó las bases del Álgebra de De Morgan, y *Boole* planteó y demostró las leyes de Boole para negar conjunciones y disyunciones.



d.

Boole sentó las bases del Álgebra de Boole, y *De Morgan* planteó y demostró las leyes de De Morgan para negar conjunciones y disyunciones.

¡Ok!

En general, una **expresión** es una fórmula en la cual se usan *operadores* (como suma, resta, comparaciones, etc.) sobre diversas variables y constantes (que reciben el nombre de *operandos* de la expresión). Son ejemplos válidos los siguientes: $3 * a + 2$, $b / c - 4$, $(7 - r) / (4 + a)$, $a > b$, $x + 2 \geq 10$.

¿Es correcta la siguiente definición?

"Una **expresión lógica** es una expresión en la cual el resultado final **es un número**"

Seleccione una:

- Verdadero
 Falso

Cuál de las siguientes expresiones lógicas es verdadera **si y sólo si** el valor de la variable **x** es 1, 2, 3, o 4?

Seleccione una:

- a.
`x == 1 and x == 2 and x == 3 and x == 4`

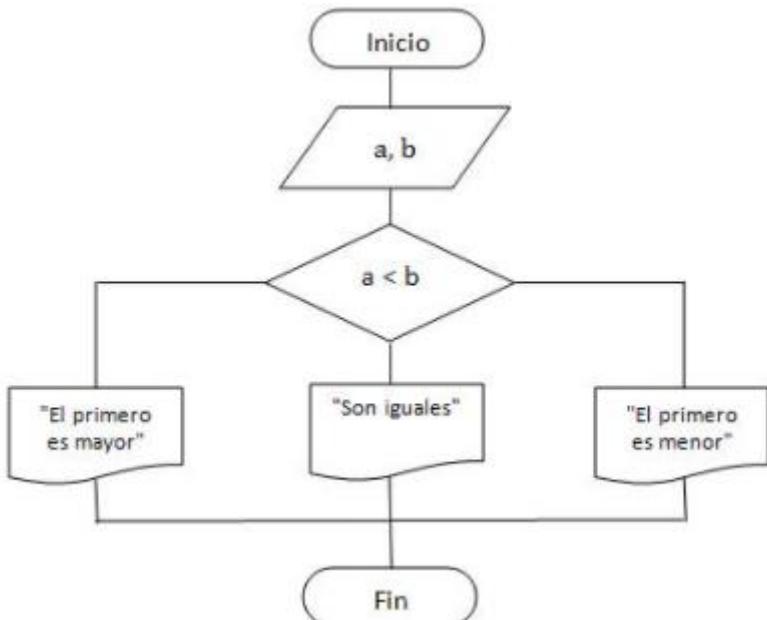
b.
`x != 1 and x != 2 and x != 3 and x != 4`

c.
`x == 1 or x == 2 or x == 3 or x == 4`

¡Ok!

- d.
`x < 0 and x > 5`

Suponga que se desea desarrollar un programa que cargue dos números y muestre un mensaje indicando si el primero es menor, igual o mayor al segundo. ¿Está bien planteado el siguiente diagrama de flujo?



Seleccione una:



a.

Está mal planteado: en la condición, se debía preguntar si $a \geq b$.



b.

Está mal planteado: el símbolo usado para la carga de datos, debió ser un rectángulo y no un paralelogramo.



c.

Está mal planteado: la condición está mal dibujada, ya que una condición no puede tener tres salidas o ramas.

¡Ok!



d.

Sí. El diagrama está correctamente planteado.

Sean las siguientes variables:

$a, b, c = 3, 10, 2$

¿Cuáles de las siguientes expresiones lógicas obtendrán un valor final **True** usando las variables y valores aquí indicados? (Observación: puede haber **VARIAS** respuestas correctas... marque **TODAS** las que considere válidas)

Seleccione una o más de una:



a.

$a == 3 \text{ or } b > 100 \text{ or } c != 2$

¡Ok!



b.

a >= b or b == 2*c or (c == 2 and a == 4)



c.

a == c and b == 10 and c != 8



d.

a != b and b != 0 and c >= 1

¡Ok!

¿Por qué motivo debe *indentarse (encolumnarse hacia la derecha)* correctamente cada rama de una instrucción condicional en Python?

Seleccione una:



a.

Para que Python pueda reconocer qué instrucciones pertenecen a cada rama. Pero aún así, la indentación sólo es obligatoria en las ramas que tengan más de una instrucción.



b.

Para que Python pueda reconocer qué instrucciones pertenecen a cada rama.

¡Ok!



c.

No es cierto que se deba indentar cada rama. La indentación sólo se sugiere por razones de claridad.



d.

Para que Python pueda reconocer qué instrucciones pertenecen a cada rama. Pero aún así, la indentación sólo es obligatoria en las ramas que tengan una y sólo una instrucción.

Esta pregunta está orientada a la aplicación de las Leyes de Morgan para negar una expresión lógica formada por conjunciones y disyunciones. Sea la siguiente expresión, sumiendo que las variables que se indican están correctamente inicializadas y con valores numéricos:

r = not(a < c or b == 5*d + 2 or not(e >= a and f != d) or d < c // 4)

¿Cuál de las siguientes expresiones lógicas es equivalente a la expresión anterior? (o sea, ¿cuál de ellas tiene la misma tabla de verdad?) Aplique las Leyes de De Morgan

con paciencia y cuidado. Recomendamos asignar valores a las variables, y probar con cada expresión que logre obtener para comparar los resultados que obtenga.

Seleccione una:



a.

```
r = a >= c and b != 5*d + 2 and e < a and f == d and d >= c  
// 4
```



b.

```
r = a >= c and b != 5*d + 2 and e >= a and f != d and d >= c  
// 4
```

¡Ok!



c.

```
r = a >= c or b != 5*d + 2 or e >= a or f != d or d >= c // 4
```



d.

```
r = a >= c and b != 5*d + 2 and e < a or f == d and d >= c //  
4
```

¿Qué tiene de malo el siguiente script en Python?

```
x1 = int(input('Primer valor: '))  
x2 = int(input('Segundo valor: '))  
  
if x1 = x2:  
    print('Son iguales')  
else:  
    print('No son iguales')
```

Seleccione una:



a.

Los mensajes que muestra en ambas ramas están al revés.



b.

No compila: usa el operador = para comparar, cuando debió usar el == (doble igual).

¡Ok! efectivamente, en Python esto provoca un error de intérprete. A diferencia de otros lenguajes, en Python el operador de asignación no puede usarse en una expresión lógica, incluso si las variables fuesen booleanas.



c.

Al ejecutar, la condición sale siempre por falso.



d.

Al ejecutar, la condición sale siempre por verdadero.

¿Cuál es el efecto del conector **and** ("y lógico") en una condición?

Seleccione una:



a.

La condición es verdadera si una y sólo una de las proposiciones es verdadera.



b.

La condición es verdadera si alguna de las proposiciones es verdadera.



c.

La condición es verdadera si todas las proposiciones son falsas.



d.

La condición es verdadera si todas las proposiciones son verdaderas.

¡Ok!

¿Cuál es el efecto del operador **or** ("o lógico") en una condición?

Seleccione una:



a.

La condición es verdadera si y sólo si todas las proposiciones son verdaderas.



b.

La condición es verdadera si y sólo si una única proposición es verdadera. Si más de una es verdadera, la salida es False.



c.

La condición es verdadera si al menos una de las proposiciones es verdadera.

¡Ok!



d.

La condición es verdadera si todas las proposiciones son falsas.

El siguiente script en Python, pretende dejar en la variable *may* el mayor de los valores contenidos en las variables *n1* y *n2*. De acuerdo a esto... ¿Hay algún problema con el script mostrado?

```
n1 = int(input('Primer valor: '))
n2 = int(input('Segundo valor: '))

if n1 > n2:
    may = n1
else: [ ]
    may = n1

print('Mayor: ', may)
```

Seleccione una:



a.

No hay ningún problema: calcula y muestra correctamente el mayor, en todos los casos.



b.

Está mal planteada la rama falsa: está asignando *n1* cuando debería asignar *n2*.

¡Ok!



c.

Está mal escrita la rama falsa: debió usar *elif* en lugar de *else*.



d.

Está mal planteada la rama verdadera: está asignando *n1* cuando debería asignar *n2*.

Suponga que se le pide desarrollar un programa que sea capaz de *elegir aleatoriamente una carta cualquiera de una (y sólo una) baraja española*. ¿Hay algún inconveniente con el programa que les mostramos aquí, o en líneas generales cumple con el requerimiento?

```
__author__ = 'Cátedra de AED'

import random

# Título principal...
print('Selección aleatoria de una carta de la baraja española...')

# Selección del número de la carta...
n = random.randint(1, 12)

# Selección del palo de la carta...
palos = 'Espada', 'Basto', 'Oro', 'Copa'
p = random.choice(palos)
```

```
# Visualización de resultados...
print('La carta seleccionada es:')
print('Palo:', p, '- Valor:', n)
```

Seleccione una:



a.

El programa no cumple correctamente con el requerimiento: a veces selecciona incorrectamente el número o valor de la carta.



b.

El programa cumple con el requerimiento sin ningún inconveniente, seleccionando siempre cartas diferentes en ejecuciones diferentes o repitiendo el mismo esquema en el mismo programa.



c.

El programa no cumple correctamente con el requerimiento: a veces selecciona incorrectamente el palo de la carta.



d.

El programa cumple con el requerimiento (aunque un inconveniente es que podría repetir la misma carta en dos ejecuciones diferentes o repitiendo el mismo esquema en el mismo programa).

¡Ok! Efectivamente, selecciona bien una carta cualquiera, pero si se ejecuta nuevamente (o se repite el mismo esquema en el mismo programa) todas las cartas vuelven a estar disponibles y podría volver a seleccionar la misma.

¿Qué hace el script que se muestra en el siguiente esquema?

```
__author__ = 'Cátedra de AED'

import random

print('Ejemplo de uso de random.random()...')
f = random.random()
i = int(f * 10)
print('El valor generado es:', i)
```

Seleccione una:



a.

Genera aleatoriamente un número en coma flotante en el intervalo [0, 1), y lo asigna en la variable *i*.



b.

Genera aleatoriamente un número entero en el intervalo [1, 10], y lo asigna en la variable *i*.



c.

Genera aleatoriamente un número entero en el intervalo [0, 9], y lo asigna en la variable *i*.

¡Ok! Efectivamente... lo primero que hace es invocar a `random.random()` y obtiene con eso un número con decimales entre 0 y 1... pero luego lo multiplica por 10 (con lo cual la coma se corre a la derecha exactamente un lugar, eliminando el cero y tomando el primer decimal para la parte entera) y finalmente trunca los decimales con la función `int()`. El resultado es un número entero de un sólo dígito, en [0, 9].

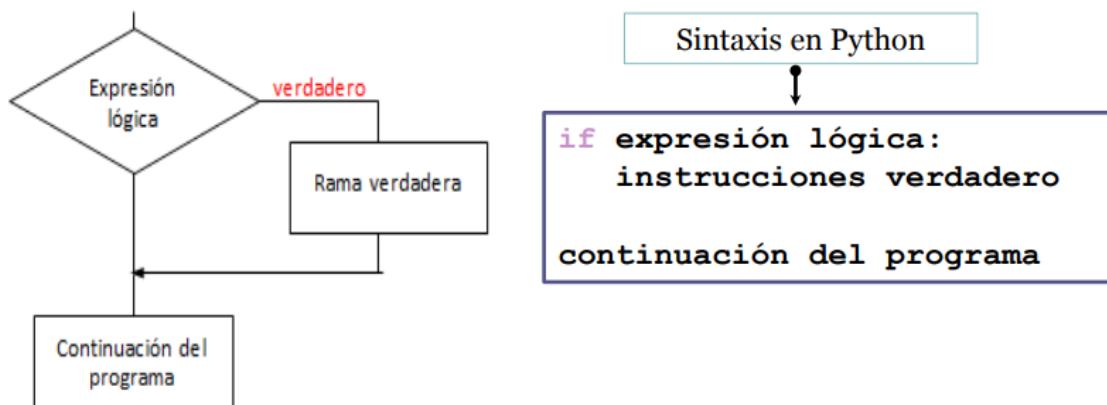


d.

Genera aleatoriamente un número en coma flotante en el intervalo [1, 10), y lo asigna en la variable *i*.

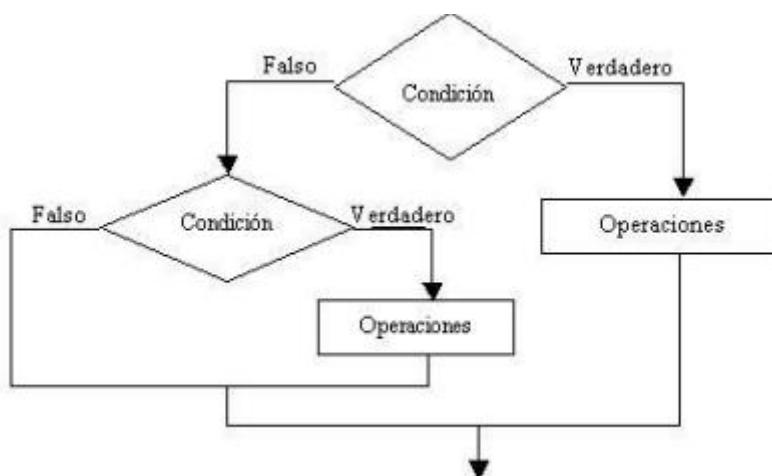
FICHA 5 Estructuras Condicionales: Variantes

- Estructura Condicional Simple: solo se conoce el accionar de una condición en el caso que la misma sea verdadera. Si la condición es falsa no se dispone información sobre los pasos a ejecutar.



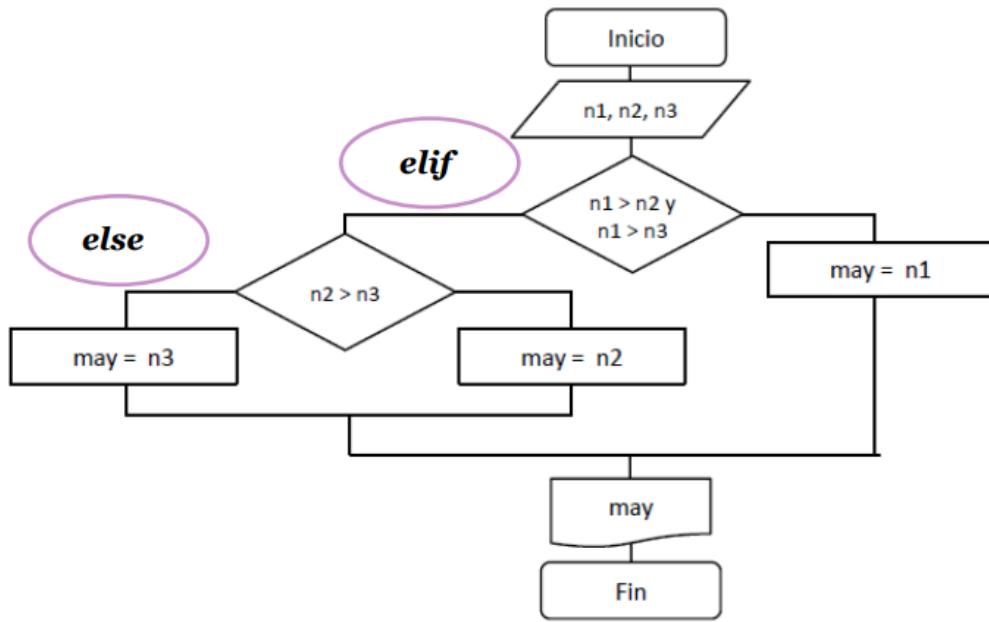
Condicional Anidadadas

- Para resolver determinados problemas es posible que la rama del verdadero y/o falso de una estructura condicional requiera plantear otra instrucción condicional.
- Es posible que a su vez cada nueva instrucción condicional incluya otras, generando lo que se conoce como anidamiento de condiciones.
- El siguiente diagrama de flujo muestra un ejemplo de anidamiento de condiciones.



Uso de elif

- Python permite plantear una variante para la instrucción condicional. Esta permite evitar el anidamiento excesivo de condiciones. El siguiente ejemplo muestra la resolución de una caso con anidamiento de condiciones.



```

# PROCESOS...
if n1 > n2 and n1 > n3:
    may = n1
elif n2 > n3:
    may = n2
else:
    may = n3

```

Anidamiento de condiciones.
Usando *elif*.

Expresiones de conteo y acumulación

- En algunos programas será necesario llevar a cabo procesos de conteo o sumarización.
- El primer caso se resuelve con el uso de una variable de conteo (contador) y en el segundo caso con una variable de acumulación (acumulador).
- En ambos casos se trata de variables que en una expresión de asignación aparecen en ambos miembros.

a = a + 1 ← Contador

b = b + x ← Acumulador

- Un contador es una variable que sirve para contar ciertos eventos del programa. Forma de un contador: $a = a + 1$
- Y se le suma el valor 1 (uno) o cualquier valor constante, cada vez que se ejecuta la expresión.
- Esta otra expresión funciona como decrementador (va decrementando de a uno) a partir de un valor original de la variable c. $c = c - 1$
- Otras variantes de expresiones de contadores:

a = a + 1

b = b - 1

c = c + 2

d = d * 4

e = e / 3

- Acumulador: es una variable que permite sumar los valores que va asumiendo otra variable u otra expresión. Forma de un acumulador: $b = b + x$

- Técnicamente un acumulador es una variable que actualiza su valor en términos de su propio valor anterior y el valor de otra variable o expresión. Ejemplo:

- Las siguientes ejemplos también representan expresiones de acumulación:

s = s + x

b = b * z

a = a - y

p = p / t

c = c + 2*x

- En Python (como en otros lenguajes) cualquier expresión de conteo o acumulación responden a la forma general siguiente:

variable = variable operador expresión

- Python permite escribir la forma anterior pero de forma resumida o abreviada:

variable operador = expresión

- En estos ejemplos se pueden notar las

<u>Forma general</u>	<u>Forma resumida</u>
<code>a = a + 1</code>	<code>a += 1</code>
<code>b = b - 1</code>	<code>b -= 1</code>
<code>c = c + 2</code>	<code>c += 2</code>
<code>d = d * 3</code>	<code>d *= 3</code>
<code>e = e / 4</code>	<code>e /= 4</code>
<code>s = s + x</code>	<code>s += x</code>
<code>b = b * z</code>	<code>b *= z</code>
<code>a = a - x</code>	<code>a -= x</code>
<code>p = p / t</code>	<code>p /= t</code>
<code>c = c + 2*x</code>	<code>c += 2*x</code>

Variables centinelas (bandera)

- Permite al programador “marcar” de alguna forma un suceso que ocurrió a lo largo de ejecución de un programa.
- Su valor se controla a lo largo del programa, de manera que cada valor se asocia con la ocurrencia o no del evento.
- Para lograr el chequeo de un suceso o evento es común es el uso de variables centinela o variables banderas (flags en inglés).
- Su tipo puede ser generalmente booleano, pero puede ser de cualquier otro tipo.

CUESTIONARIO

¿En qué famoso museo de la ciudad de Londres (Inglaterra) se ha construido y se expone al público una *Analytical Engine*, la máquina diseñada por *Charles Babbage* a principios del siglo XIX y que es la precursora de las modernas computadoras? [El museo construyó la máquina usando materiales que estaban disponibles en la época en que Babbage la diseñó...]

Seleccione una:



- a.
Museo de Cera



- b.
Museo de Guerra



- c.
Museo Británico



- d.
Museo de Ciencias

¡Ok!

¿Cuántas condiciones **como mínimo** necesita en un programa en Python para encontrar el menor valor de entre tres variables (sin usar la función *min()* de la librería estándar)?

Seleccione una:



- a.
Una condición.



- b.
Dos condiciones.

¡Ok! Puede plantearlo de varias formas, pero le quedarán dos condiciones al final...



- c.
Cuatro condiciones.



- d.
Tres condiciones.

¿Qué hace siguiente script en Python?

```
a = int(input('Ingrese limite izquierdo: '))
b = int(input('Ingrese limite derecho: '))
if a > b:
    a, b = b, a

x = int(input('Ingrese el valor a chequear: '))
if a <= x <= b:
    print('Ok...')
else:
    print('Revise...')
```

Seleccione una:



a.

Determina si el valor x está contenido en el intervalo $[a, b]$. Muestra el mensaje "Ok..." si es así, o muestra "Revise..." si no es cierto.

¡Ok!



b.

Determina si el valor x es mayor que los valores a y b . Muestra "Ok..." si es cierto, o "Revise..." si no es cierto.



c.

Determina si x es igual al promedio de los valores a y b . Muestra el mensaje "Ok..." si es cierto, o el mensaje "Revise..." si no es así.



d.

Ordena los valores a , b , x y muestra el valor "Ok..." al final si quedaron ordenados de menor a mayor, o bien muestra "Revise..." si quedaron de mayor a menor.

¿Qué es lo que hace el siguiente script?

```
__author__ = 'Cátedra de AED'

a = int(input('A: '))
b = int(input('B: '))

may = a
if b > a:
    may = b

print('El mayor es:', may)
```

Seleccione una:



a.

Sólo si el valor de b es mayor que el de a , deja la variable may con el valor de b . Si el mayor es a , la variable may queda con valor indefinido.



b.

Lanza un error de intérprete: la instrucción condicional debió tener un *else* para iniciar la rama falsa.



c.

Lanza un error al ejecutar: las variables *a* y *b* no están definidas cuando se ejecuta la condición.



d.

Determina el mayor entre las variables *a* y *b*, y deja el valor mayor en la variable *may*.

¡Ok! Efectivamente, comienza suponiendo que el mayor es *a* y asigna ese valor en *may*. Si al preguntar si $b > a$ la condición es falsa, el mayor era efectivamente *a* y no hay que cambiar el valor de *may*. Pero si la condición es cierta, el mayor entonces es *b*, y entonces sólo en este caso se cambia el valor de *may*.

¿Cuál de los siguientes es el nombre del sistema de cifrado de mensajes usado por los alemanes en la Segunda Guerra Mundial, para cuyo descifrado el Ing. Thomas Flowers diseñó la famosa máquina Colossus?

Seleccione una:



a.

Enigma



b.

Vigenère



c.

Cesar



d.

Lorenz SZ42

¡Ok!

¿Cuál es el nombre del matemático, filósofo e informático teórico en cuyo honor la Association for Computing Machinery (ACM) instituyó un premio anual con su nombre, considerado como el equivalente al Nobel en las Ciencias de la Computación?

Seleccione una:



a.

Charles Antony Hoare



b.

Alan Turing

¡Ok!



c.

Adi Shamir



d.

James Gosling

¿Qué hace el siguiente segmento de programa?

```
a = int(input('Primer número: '))
b = int(input('Segundo número: '))

if a < b:
    print('El primer número es el menor')
print('El primer número es el mayor')
```

Seleccione una:



a.

Si a es mayor que b , muestra sólo el mensaje "*El primer número es menor*"



b.

Si a es menor que b , muestra dos mensajes: "*El primer número es menor*" y luego: "*El primer número es mayor*".

¡Ok! Obviamente, lo que hace este script es incorrecto... pero sólo se le pidió indicar qué hacía, y no si su salida era correcta...



c.

Si a es menor que b , muestra un único mensaje: "El primer número es menor".



d.

Si a es mayor que b , no muestra un ningún mensaje.

¿Qué hace el siguiente script Python?

```
a = int(input('A: '))
b = int(input('B: '))
c = int(input('C: '))
d = int(input('D: '))
```

```
m1 = a
if b > a:
    m1 = b
```

```
m2 = c
if d > c:
    m2 = d
```

```
if m1 > m2:
    print(m1)
else:
    print(m2)
```

Seleccione una:



a.

Muestra el menor de los valores almacenados en las variables a, b, c, d.



b.

Muestra el mayor de los valores almacenados en las variables a, b, c, d.

¡Ok!



c.

Muestra los dos mayores entre los valores a, b, c, d.



d.

Muestra los cuatro valores a, b, c, d en orden de menor a mayor

Analice el siguiente script básico en Python:

```
z = 3
x = 0
x = x + 1
x = x + 3
x = x + z
print('x:', x)
```

¿Cuál es el valor final almacenado en la variable x que será mostrado al ejecutar el script anterior?

Seleccione una:



a.

El valor 7.

¡Ok!



b.

El valor 3.



c.

El valor 0.



d.

El valor 6.

Las instrucciones que siguen son expresiones de acumulación o conteo en Python, escritas en la forma general. Seleccione para cada una de ellas, su correspondiente forma resumida:

`x = x / 3`

Respuesta 1 ▾

`x = x + pow(y, 0.5)`

Respuesta 2 ▾

`x = x - 5*d`

Respuesta 3 ▾

`x = x + p`

Respuesta 4 ▾

`x = x % 2`

Respuesta 5 ▾

`x = x * 6`

Respuesta 6 ▾

FICHA 11 Módulos y Paquetes

- **Parametrización:** mecanismo que favorece el desarrollo de funciones genéricas y reutilizables. Ejemplo:

- Python puede aceptar y manejar un número variable de parámetros cuando se invoca un función. Ejemplo: `m1 = max(3,5,6) m2 = max(2,5,6,7,3,1)`

- El programador puede implementar funciones con esta particularidad, Python provee tres mecanismos:
 - Parámetros con valores por defecto (default).
 - Parámetros con palabra clave.
 - Lista de parámetros con longitud arbitraria.

En Python es posible asignar valores por defecto a los parámetros formales de una función. Esto implica que la función puede ser invocada con menos parámetros de lo que espera.

Este tratamiento especial de parámetro se formaliza: def nombre_funcion(param1,param2 = valor) #Cuerpo de la función

```
def ordenar(n1, n2,  
ascendent=True):  
    # se asume ascendent True  
    first, second = n2, n1  
    if n1 < n2:  
        first, second = n1, n2  
  
    # si ascendent = False,  
    #invertir los valores...  
    if not ascendent:  
        first, second = second, first  
    return first, second
```

*Si al invocar a ordenar() no se envía el 3 parámetro, se asume el valor True (defecto).
* Si se envía el tercer parámetro entonces recubre el valor por defecto.

```
def test():  
    a = int(input('Ingrese 1: '))  
    b = int(input('Ingrese 2: '))  
    # orden ascendente...  
    men, may = ordenar(a, b)  
  
    print('Menor:', men)  
    print('Mayor:', may)  
  
    # orden descendente...  
    may, men = ordenar(a, b, False)  
  
    print('Menor:', men)  
    print('Mayor:', may)  
  
# Activo función principal.-  
test()
```

- Los parámetros formales que NO tienen un valor por defecto, se designan como parámetros posicionales, y es obligatorio enviar los parámetros actuales que se corresponden a ellos .

may, men = ordenar(a) # Incorrecto 😞...

men, may = ordenar() # Incorrecto Λ...

- Los parámetros posicionales en la cabecera de la función deben definirse antes que los que llevan un valor por default:

incorrecto...

def ordenar(n1, ascendent=True, n2):

- Otras consideraciones a tener en cuenta y vale la pena analizar:

```
may, men = ordenar(a, b, False)      # correcto...  
men, may = ordenar(c, d)              # correcto...  
men, may = ordenar(c, d, True)       # correcto...  
men, may = ordenar(a, False)         # Atención...
```

```
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() # Error
    datos('Luigi') # OK.
    datos('Camila', 'Argentina', 'Mujer') # OK.
    #
    datos('Bruno', 'Varon', 'Italia') # Genera ambigüedad!!
test()
```



- Los parámetros por defecto (default) no pueden saltarse, para evitar este problema al invocar la función se puede nombrar directamente a un parámetro por su nombre (o palabra clave)

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

```
def test():
    datos('Luigi', país='Inglaterra') # OK.
    datos(nombre='Luigi', país='Inglaterra') # OK
    datos('Tomás', 'Inglaterra', sexo='Mujer') # OK
    datos('Alejandro', pais='Argentina', sexo='Varón') # OK
    #
    datos('Claudio', pais='Uruguay', 'Mujer') ! # Error: luego de palabra clave, no puede ser explícito.
    # Error: luego de palabra clave, no puede ser explícito.
test()
```



- Python usa en forma masiva el mecanismo de parámetros con valor por defecto (default), como el mecanismo de selección de parámetros por palabra clave en diferentes de sus funciones predefinidas.

- Un ejemplo es la función print(). Esta dispone de dos parámetros por defecto que son seleccionados por palabras claves: sep y end.

sep: se usa para indicar a la función que carácter se debe usar para separar las cadenas, por default es un espacio en blanco.

end: se usa para indicar con carácter termina la visualización, que por defecto es un salto de línea.

- Por este motivo la función, la salida producida es:

```
x, y = 10, 20
print('Valor x:', x)      → Valor x: 10
print('Valor y:', y)      → Valor y: 20
```

- Estos caracteres pueden cambiar llamando a la función y seleccionando esos parámetros por palabra clave.

```
x, y = 10, 20
print('Valor x:', x, sep='--> ', end='\n\n')
print('Valor y:', y)
↓
Valor x:--> 10
Valor y: 20
```

- Este tratamiento especial de parámetro permite que una función acepte un número arbitrario de parámetros.
- Los parámetros enviados entran a la función empaquetados en una tupla.
- Este parámetro se escribe colocando un *nombre detrás del último parámetro posicional que la función haya definido.

Un ejemplo de parámetros de longitud arbitaria se formaliza en la cabecera de la función como se muestra a continuación: def funcion(param1,param2, *args)

```

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)

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

    # script principal.....
test()

```

Módulos en Python

- 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 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

Ejemplo de definición de un módulo en Python:

```

# "soporte.py"...
def menor(n1, n2):
    if(n1 < n2):
        return n1
    return n2

def factorial(n):
    f = 1
    for i in range(2,n+1) :
        f*=i
    return f

```

```

# test01.py
import soporte
def test():
    a = int(input("Cargue el primer número: "))
    b = int(input("Cargue el segundo número: "))
    #invocación a las funciones del módulo "soporte"
    m = soporte.menor(a,b)
    # si una función se va a usar mucho, se puede
    # la referenciar con un identificador local...
    men = soporte.menor
    c = men(3,5)
    print("El menor es: ", m)
    print("El segundo menor es: ", c)

# script principal...
test()

```

Variantes de import

- import nombre-módulo: introduce el nombre del módulo en el contexto del script, pero no el nombre de las funciones definidas en él. Para invocar a una de sus funciones debe usarse el operador (.). Es decir: nombre-módulo + punto + función a invocar

```

import soporte

m = soporte.menor(5,6)

f = soporte.factorial(3)

```

- 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 f= menor(6,5)

Variables Especiales

- Todo módulo en Python tiene una tabla de símbolos propia, que contiene las variables y funciones definidos en el módulo. Esta tabla se carga en memoria al momento que el módulo es accedido por la instrucción import o from-import
 - Un módulo además incluye algunas **variables globales especiales**. Ejemplo: la variable **author**, usada para contener el nombre de programador o equipo que desarrolló el programa.
 - Otra variable especial de uso muy común se denomina **name**, que es asignada directamente con el nombre del módulo.
 - El siguiente ejemplo muestra la forma de acceder a las variables **author** y **name**.

```

import soporte
# Archivo prueba04.py

def test():
    print('Factorial de 4:', soporte.factorial(4))
    print('Autor modulo :', soporte.__author__)
    print('Nombre del modulo:', soporte.__name__)

test()

```

```

Factorial de 4: 24
Autor modulo: Cátedra de
AED
Nombre del modulo: soporte

```

- Todo archivo de código fuente en Python (extensión .py) es un módulo. Si contiene un script principal se dice que ejecutable (porque al ejecutarse se lanzará el script principal). Pero si NO tiene un script principal, al ejecutar no ocurrirá nada e informa: Process finished with exit code 0
- Si un módulo ejecutable (contiene un script) es importado por otro, al momento de la importación el módulo se cargará y se ejecutará el script de ese módulo. • Para evitar esto, hay que conocer que la variable __name__ de un módulo se inicializa en forma automática, pero el valor inicial depende de cómo fue cargado el módulo.

-
1. Si el módulo fue importado (con import o con from import), entonces __name__ se inicia con el nombre del módulo.
 2. Si el módulo fue ejecutado en forma directa (no importado), entonces __name__ se inicializa con la cadena "__main__"
-

- Para controlar si un módulo fue importado o ejecutado en forma directa, se debe agregar esta condición para su script principal.

```

if __name__ == "__main__":
    # se pidió ejecutar ESTE módulo. Script principal...

```

- Si un módulo NO tiene un script principal, no es necesario incluir esta condición en el módulo. Pero si el módulo TIENE un script principal (es ejecutable) debería tener esta condición.
- Esta convención seguiremos de ahora en adelante es la de incluir esta condición para contener al script principal de nuestros programas, aún cuando no haya a la vista ningún otro módulo.

Librería estándar de Python

- Python dispone de un conjunto de módulos ya implementados y listos para usar.

Módulo Estándar	Contenido General
math	Funciones matemáticas típicas (logarítmicas, trigonométricas, etc.)
random	Funciones para el trabajo con números aleatorios y decisiones aleatorias.
datetime	Funciones para el trabajo con fechas y horas.
os	Interfaz de acceso a funciones del sistema operativo.

Librería estándar de Python

- Python provee un conjunto de funciones internas que están disponibles sin necesidad de importar ningún módulo. Algunas de ellas son:

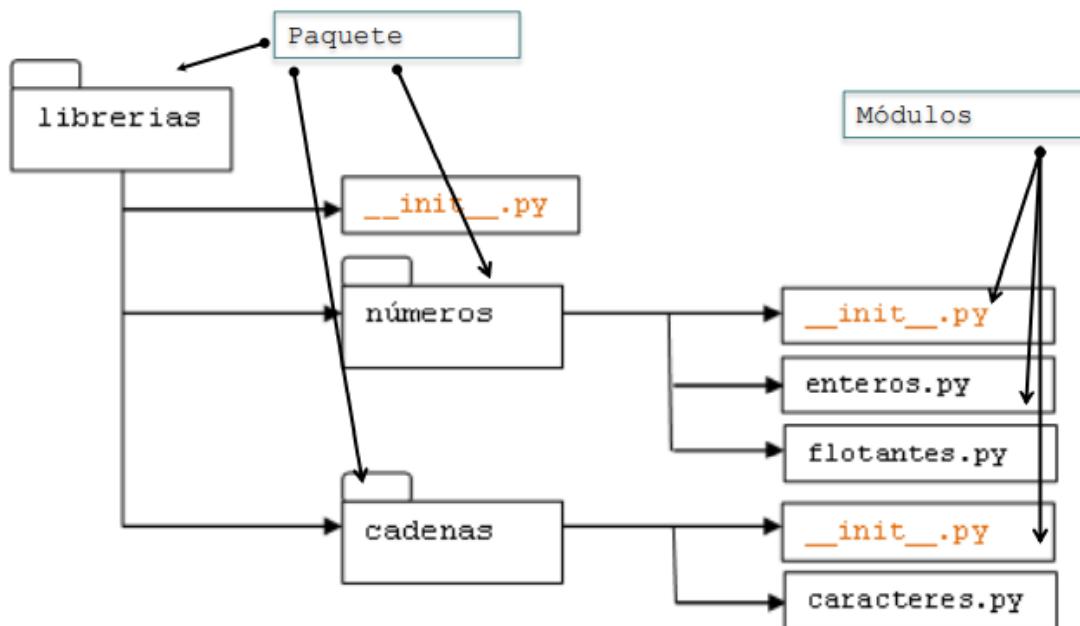
Función	Descripción
abs(x)	Retorna el valor absoluto del parámetro X.
bin(x)	Convierte el entero x en una cadena binaria.
len(s)	Retorna la longitud del objeto s tomado como parámetro (una string, una lista, una tupla o un diccionario).
int(x)	Convierte el parámetro x a un número entero.

Definición de paquetes en Python

- Un paquete (o package) en Python es una forma de organizar y estructurar carpetas de módulos.
- La forma de acceso a cada módulo es mediante el recurso del operador punto para especificar la ruta de acceso y el nombre de cada módulo. Ejemplo: Si se tiene un paquete llamado módulos y dentro de él se incluye un módulo llamado funciones y con ese nombre deberá ser accedido por la sentencia import.
- La ventaja de usar paquetes permite evitar posibles ambigüedades o conflictos de nombres que se pueden generar si distintos programadores aportan diferentes módulos que incluyen funciones llamadas de igual forma.
- La creación de un paquetes de módulos no conlleva un proceso complicado en Python. No todos los IDEs para Python proveen esta funcionalidad.

- El IDE PyCharm si permite la creación de paquetes de módulos. Sobre un proyecto ya creado, se presiona botón derecho, seleccione New y luego la opción “Python Package”.
- La carpeta creada que representa el nombre del paquete contiene un archivo `__init__.py` posiblemente vacío o conteniendo la variable global `__author__`.

Definición de paquetes en Python



- Módulo enteros que pertenece al paquete librerías-números. Tiene el siguiente contenido

```
# una función para retornar
#el menor entre dos números...
def menor(n1, n2):
    if n1 < n2:
        return n1
    return n2
```

```
# una función para calcular el
#factorial de un número...
def factorial(n):
    f = 1
    for i in range(2,n+1):
        f *= i
    return f
```

- Módulo flotantes que pertenece al paquete librerías-números. Tiene el siguiente contenido:

```

__author__ = 'Cátedra de AED'

# una función para obtener el promedio entre los
# parámetros
def promedio (x1, x2, *x):
    suma = x1 + x2
    conteo = 2

    n = len(x)
    if n != 0:
        for i in range(n):
            suma += x[i]
        conteo += n

    return suma / conteo

```

- Módulo `cadenas.caracteres` que pertenece al paquete `librerías/cadena`. Tiene el siguiente contenido

```

__author__ = 'Cátedra de AED'

# funciones genéricas para el tratamiento de cadenas..

def es_vocal(car):
    if car in 'aeiouAEIOU':
        return True
    return False

def es_dígito(car):
    if car in '0123456789':
        return True
    return False

```

- Para acceder a los módulos de un paquete (package), se usan las directivas `import` o `from-import`, pero agregando el nombre del paquete y la ruta de subcarpetas hasta llegar al módulo, separando cada nombre en esa ruta con punto. Ejemplo con directiva `import`

```
import librerias.numeros.flotantes
import librerias.cadenas.caracteres

def test():
    p = librerias.numeros.flotantes.promedio(2.34, 4, 5.34)
    print("Promedio: ", p)
    if librerias.cadenas.caracteres.es_vocal("a"):
        print("Carácter es un vocal.")
    else:
        print("Carácter NO es un vocal.")

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

- Ejemplo para el acceso de los módulos de un paquete (package) con directiva from – import

```
from librerias.numeros import flotantes, enteros
from librerias.cadenas.caracteres import es_vocal

def test():
    p = flotantes.promedio(2.34, 4, 5.34)
    print("Promedio: ", p)
    m = enteros.menor(4, 6)
    print("El menor es: ", 4)
    if es_vocal("a"):
        print("Es vocal.")
    else:
        print("No es vocal.")

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

CUESTIONARIO

Suponga la siguiente cabecera de una función cuyo bloque de acciones es irrelevante a los efectos de la pregunta:

```
def ejemplo(a, b=0, c='Desconocido', d='Ingeniero'):
    # el bloque de acciones no importa ahora...
```

¿Cuales de las siguientes invocaciones a esta función son correctas y activan la función sin producir un error de intérprete? (Observación: más de una respuesta puede ser correcta. Marque todas las que considere válidas... y tómese su tiempo para pensar y probar cada respuesta posible...)

Seleccione una o más de una:



a.

```
ejemplo(40, 'Juan', 'Abogado')
```

Correcto. Efectivamente, el primer parámetro es obligatorio (y en este caso es un número). El segundo y el tercero quedan valiendo las cadenas "Juan" y "Abogado". Y el restante queda valiendo su valor default [*Valores finales de los parámetros: a = 40, b = "Juan", c = "Abogado", d = "Ingeniero"*] Si este resultado le parece extraño, ¡recuerde que los parámetros se toman y se asignan por orden de aparición, y que Python es un lenguaje de tipado dinámico!



b.

```
ejemplo(32, 20)
```

Correcto. Efectivamente, el primer parámetro es obligatorio (y en este caso es un número), y el segundo es opcional dado que admite un valor default. Los otros dos quedan valiendo valores default [*Valores finales de los parámetros: a = 32, b = 20, c = "Desconocido", d = "Ingeniero"*]



c.

```
ejemplo('Medico')
```

Correcto. Efectivamente, un solo parámetro es obligatorio en esta función y por lo tanto el único valor enviado ('Medico') se asigna en ese parámetro (la variable *a*). El resto toma sus valores por default [*Valores finales de los parámetros: a = Medico, b = 0, c = "Desconocido", d = "Ingeniero"*] Si este resultado le parece extraño, ¡recuerde que los parámetros se toman y se asignan por orden de aparición, y que Python es un lenguaje de tipado dinámico!



d.

```
ejemplo()
```

Suponga que se quiere desarrollar una función que tome como parámetro una secuencia (un string, una lista, una tupla, o cualquier otro tipo de colección que permita acceso mediante índices en Python) y que proceda a ordenar esa colección con diversos algoritmos conocidos, en forma ascendente o descendente a elección de quien invoca a la función.

En ese contexto, considere la siguiente definición para esa función, en la cual no es relevante su bloque de acciones a los efectos de la pregunta:

```
def ordenar(lista, ascendente=True, algoritmo='Quicksort'):
    # el bloque de acciones no importa ahora...
```

¿Cuáles de las siguientes invocaciones a esta función son correctas, y no provocarán un error de intérprete? (Observación: más de una respuesta puede ser válida. Marque todas las que considere adecuadas... y de nuevo: tómese su tiempo!!!)

Seleccione una o más de una:



a.

```
ordenar('azbycx', algoritmo='Shellsort')
```

Correcto. Efectivamente, el primer parámetro es obligatorio en esta función. El segundo queda con valor default. Y el tercero es seleccionado por palabra clave [Valores finales de los parámetros: lista = "azbycx", ascendente = True, algoritmo = "Shellsort"]



b.

```
ordenar('abcdef', ascendente=False, 'Insertionsort')
```



c.

```
ordenar(lista=(1,2,3), algoritmo='Heapsort', ascendente=False)
```

Correcto. Efectivamente, el primer parámetro es obligatorio en esta función, y no sólo eso sino que en este caso se lo está accediendo por palabra clave y queda valiendo una tupla compuesta de números. El segundo y el tercero son seleccionados por palabra clave, sin importar si se los está accediendo en orden diferente al de su declaración. [Valores finales de los parámetros: lista = (1,2,3), ascendente = False, algoritmo = "Heapsort"]



d.

```
ordenar('abcdef', False, metodo='Bubblesort')
```

La siguiente función se propone tomar como parámetro el nombre de un empleado y los importes de los últimos sueldos que percibió para luego mostrar un listado que incluya el nombre recibido y el promedio de sus sueldos:

```
def procesar(nombre, *importes):
    print('Nombre del empleado:', nombre)

    p = 0
    n = len(importes)
    if n != 0:
        s = 0
        for imp in importes:
            s += imp
        p = s / n
    print('Sueldo promedio:', p)
```

¿Cuáles de las siguientes invocaciones son correctas, en el sentido de ejecutarse sin problemas y mostrar los resultados pedidos sin provocar una interrupción del programa? (Observación: más de una respuesta puede ser válida, por lo que marque todas las que considere oportunas).

Seleccione una o más de una:



a.

```
procesar('Pedro', 1234.56, 2345.45)
```

Correcto. La tupla **importes** tiene dos valores numéricos de tipo *float*, que son procesados correctamente, entregando un sueldo promedio de 1208.1866666666667



b.

```
procesar('Carlos', '1000', '2000', '3000')
```



c.

```
procesar('Laura', 1000, '2000', True)
```



d.

```
procesar('Luis')
```

Correcto. En este caso, la tupla **importes** está vacía, y se muestra un sueldo promedio igual a cero, correctamente.

¿Cuál de las siguientes afirmaciones es **incorrecta** en relación al concepto de **módulo** en Python, y/o a elementos asociados al uso de módulos en Python?

Seleccione una:



a.

Un módulo siempre puede ser importado desde otro módulo, mediante instrucciones *import* o *from import*.



b.

Cada vez que un módulo es importado en un programa, su contenido es vuelto a compilar por el intérprete.

¡Correcto! Esta es justamente la afirmación incorrecta pedida: no es cierto que un módulo se compila nuevamente cada vez que se lo importa. Sólo se compila la primera vez, y esa precompilación se almacena en la carpeta `_pycache_` del proyecto. Con esto se gana tiempo de ejecución, al no tener que compilar una y otra vez un módulo cada vez que un programa lo importa o lo accede.



c.

Un módulo en Python es cualquier archivo con extensión `.py` (código fuente que contenga funciones, definiciones generales, clases, variables, etc.)



d.

Cuando se usa una instrucción import, Python busca primero si existe un módulo estándar cuyo nombre coincide con el del módulo importado. Si no lo encuentra, busca entonces en la lista de carpetas indicada por la variable sys.path.



e.

Un módulo pensado para ser ejecutado en forma directa debería contener un script de control de la forma `if __name__ == '__main__'` para evitar que al ser importado se ejecute en forma inconveniente su posible script principal.



f.

Un módulo en Python puede tener elementos *docstring* que documenten su uso y su contenido.

Suponga las siguientes instrucciones `import` en un programa (note que todos los módulos nombrados en esos `import` pertenecen a la librería estándar de Python):

```
import math  
from re import split  
from random import random
```

¿Cuáles de las siguientes invocaciones a funciones son **correctas**, considerando los `import` que se acaban de mostrar? (Observación: más de una respuesta puede ser válida. Marque todas las que considere oportunas)

Seleccione una o más de una:



a.

```
# observacion: la funcion sqrt() (raiz cuadrada) pertenece al  
modulo math  
x = math.sqrt(4)
```

Correcto. Hay un import que introduce el nombre del módulo math en este programa, y a partir de allí cada función de ese módulo debe ser nombrada con el prefijo "**math.**"



b.

```
# observacion: la funcion random() pertenece al  
modulo random  
z = random.random()
```



c.

```
# observacion: la funcion log2() (logaritmo en base 2)  
pertenece al modulo math  
lg = log2(8)
```



d.

```
# observación: la función split() (partir en subcadenas)
pertenece al modulo re
y = split("\L+", "La ola de la vida")
```

Correcto. Hay una instrucción de importación que específicamente introduce el nombre de la función `split()` en este script, por lo cual no debe usarse el prefijo "`re`."

En la columna de la izquierda, se nombran algunos de los módulos que existen en la librería estándar de Python. Seleccione el uso o aplicación de cada uno de estos módulos.

urllib.request	Respuesta 1 Recuperación de datos desde un url.
datetime	Respuesta 2 Manipulación de fechas y horas
os	Respuesta 3 Acceso a funciones del sistema operativo.
doctest	Respuesta 4 Validaciones de tests incluidos en strings de documentación.
sys	Respuesta 5 Variables de entorno y acceso a parámetros de línea de órdenes
xml.dom - xml.sax	Respuesta 6 Parsing de documentos XML.
re	Respuesta 7 Reconocimiento de patrones

Suponga la definición del siguiente módulo en Python, **tal como se muestra**, y suponga también que este módulo se ha almacenado en el archivo "`cadenas.py`:

```
def mensaje(m) :
    print('El mensaje es:', m)

def invertir(m) :
    print('El mensaje (invertido) es:')

    n = len(m)
    for i in range(n-1, -1, -1):
        print(m[i], end='')

def test():
    cadena = input('Ingrese un mensaje: ')
    mensaje(cadena)
    invertir(cadena)

test()
```

Suponga ahora que se define otro módulo (almacenado en el archivo "`prueba.py`" y en la misma carpeta que el módulo "`cadenas.py`") cuyo contenido es el siguiente, **tal como se muestra**:

```
import cadenas

def principal():
    cad1 = 'Universidad Tecnologica Nacional'
    cadenas.mensaje(cad1)
    cadenas.invertir(cad1)

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

¿Cuál de las siguientes afirmaciones describe correctamente lo que ocurrirá si se pide ejecutar el contenido del módulo "prueba.py"?

Seleccione una:



a.

El módulo prueba.py está importando al módulo cadenas.py y como este último **no** incluye un chequeo de la forma `if __name__ == '__main__'`, entonces al ejecutar prueba.py se ejecutará primero la función `cadenas.test()` y luego la función `prueba.principal()`.

¡Correcto!



b.

El módulo prueba.py está importando al módulo cadenas.py y como este último **no** incluye un chequeo de la forma `if __name__ == "__main__"`, entonces al ejecutar prueba.py se ejecutará primero la función `prueba.principal()` y luego la función `cadenas.test()`.



c.

El módulo prueba.py está importando al módulo cadenas.py y como prueba.py incluye un chequeo de la forma `if __name__ == '__main__'`, entonces al ejecutar prueba.py se ejecutará solamente la función `prueba.principal()`.



d.

El módulo prueba.py está importando al módulo cadenas.py y como este último **no** incluye un chequeo de la forma `if __name__ == "__main__"`, entonces al ejecutar prueba.py se producirá un error de intérprete cuando se intente cargar el módulo `cadenas.py`.

Para cada una de las variables predefinidas de uso global de Python de la primera columna, seleccione la definición que le corresponde o que mejor se le ajuste.

<code>__author__</code>	Respuesta 1	El nombre del autor del elemento.	<input type="button" value="▼"/>
<code>__name__</code>	Respuesta 2	El nombre de un módulo o el valor literal ' <code>__main__</code> '.	<input type="button" value="▼"/>

<u>doc</u>	Respuesta 3	Todos los docstrings que contiene el elemento.
<u>all</u>	Respuesta 4	Los elementos a importar desde un paquete.

Suponga que se tiene un paquete en Python llamado **soporte**, y que ese paquete contiene cuatro módulos llamados respectivamente *modelo*, *persistencia*, *interfaz* y *excepciones*. ¿Cuáles de las siguientes **son correctas** en relación al archivo `__init__.py` del paquete **soporte**? (Más de una puede ser válida... marque todas las que considere apropiadas)

Seleccione una o más de una:

a.

El archivo `__init__.py` del paquete **soporte** puede contener una asignación con los nombres de todos o algunos de los módulos que posee, en la variable `_all_`.

Correcto... sabemos que se puede usar la variable `_all_` para asignar en ella los nombres de los subpaquetes y/o módulos que queremos que se importen si se hace un **from soporte import ***.

b.

El archivo `__init__.py` del paquete **soporte** debe estar presente en la carpeta de ese paquete, para que Python lo reconozca como un paquete válido.

Correcto... justamente, la presencia de ese archivo es lo que hace que Python considere a la carpeta como un package.

c.

El archivo `__init__.py` del paquete **soporte** puede dejarse vacío. No importa si el paquete tiene subpaquetes y módulos o no.

Correcto... dejar vacío el archivo `__init__.py` es válido... aunque a veces podría no tener mucha utilidad.

d.

El archivo `__init__.py` del paquete **soporte** no puede dejarse vacío. Debe asignar los nombres de los módulos que posee, en la variable `_all_`.

Suponga que se tiene un paquete llamado *test* en Python, y que ese paquete contiene tres subpaquetes llamados *prueba1*, *prueba2* y *prueba3*. Suponga además que archivo `__init__.py` del paquete *test* contiene solo el siguiente *docstring* (y ninguna otra asignación o script adicional):

```
"""El paquete contiene subpaquetes para operaciones de
testing de un sistema.
Lista de subpaquetes incluidos:
```

```
:prueba1: Contiene un modulo con funciones para testear  
operaciones de input/output  
:prueba2: Contiene dos modulos con funciones para testear  
manejo de excepciones  
'''
```

¿Hay algún problema con este bloque, en relación al respeto de las convenciones de documentación *docstring*? (Marque la respuesta que mejor describa la situación)

Seleccione una:



a.

El problema es el propio bloque en sí mismo: un paquete no lleva documentación *docstring*.



b.

Hay un par de problemas: debería haber una línea en blanco después de la primera línea del bloque (la descripción de los subpaquetes debería aparecer luego de esa línea en blanco); y además, faltaría la descripción del subpaquete *prueba3*.

Correcto. Las convenciones generales sugieren la línea en blanco faltante, y las convenciones para describir un paquete sugieren que se documente brevemente todo subpaquete que el paquete contenga (salvo que algún subpaquete se haya excluido por no haberlo asignado en la variable *_all_*... ¡que no es el caso!)



c.

No hay ningún problema.



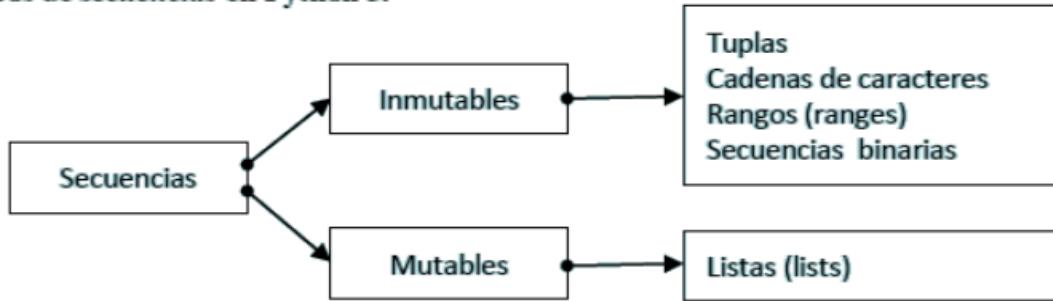
d.

Hay un solo y único problema: debería haber una línea en blanco después de la primera línea del bloque (la descripción de los subpaquetes debería aparecer luego de esa línea en blanco)

FICHA 12 Arreglos Unidimensionales

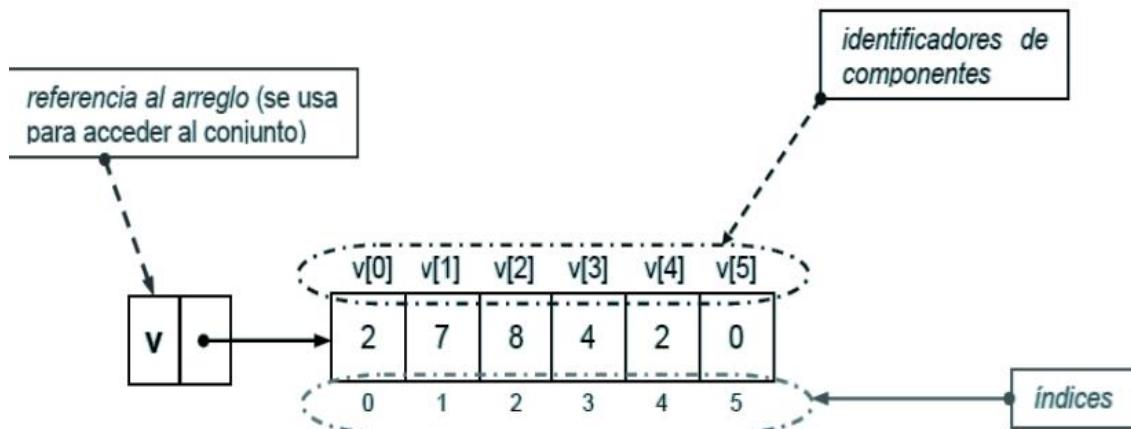
- Estructura de datos: es una variable que puede contener varios valores a la vez.
- Python provee ciertas estructuras de datos en forma de secuencias.
- La ventaja de usar estructuras de datos para el desarrollo de programas, es la posibilidad de manejar un volumen significativo de datos y/o resultados.

Tipos de secuencias en Python 3.



- Arreglo unidimensional: es una estructura de datos básica que almacena un conjunto de valores, de manera tal que cada componente individual es identificado por un número denominado índice.
- El uso de índices permite el acceso, uso y/o modificación de cada componente en forma individual.
- La cantidad de índices que se requieren para acceder a un elemento individual, se llama dimensión del arreglo.
- Los arreglos unidimensionales se denominan así, porque requieren solo un índice para acceder a un componente.

Figura 1: Esquema básico de un arreglo unidimensional.



Conceptos

- Python provee un tipo de secuencia designado como la lista o (list) , que permite a un programador gestionar un arreglo en forma completa.
 - Por su similitud que existe entre el concepto de arreglo unidimensional y el concepto de vector en Algebra, suelen también llamarse vectores.
 - En Python es posible la definición de un arreglo que puede contener valores de distintos tipos.
- `datos = ['Juan', 35, True, 5678.56]`
- En el caso de esta definición de arreglo (lista) el programador deberá tener en cuenta en que posición almacenó un determinado tipo de dato para evitar un procesamiento incorrecto.

Creación de un arreglo unidimensional

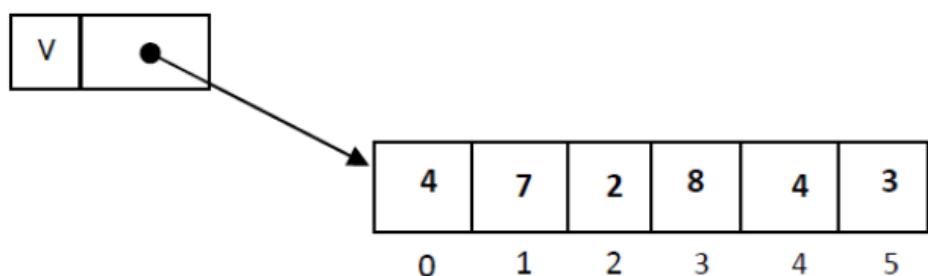
- Para crear un arreglo unidimensional “v” vacío, se debe definir una variable de tipo list en cualquiera de las dos forma que se muestra a continuación:

v = [] o

v = list()

- Si se conoce los valores de antemano los valores que se desea almacenar, se puede crear enumerando sus componentes. v = [4, 7, 2, 8, 4, 3]
- La última instrucción anterior crea un arreglo de seis componentes (por enumeración), inicializa cada casilla de ese arreglo con los valores enumerados entre los corchetes, y retorna la dirección la dirección del arreglo.

Figura 2: Un arreglo con seis componentes de tipo int.



- Al igual que las cadenas y tuplas, el contenido completo de una variable list puede mostrarse directamente en la consola estándar mediante una simple invocación a la función print().

- La siguiente secuencia de instrucciones muestra contenido de un arreglo: v = [4, 7, 2, 8, 4, 3]
print('Contenido del arreglo:')

```
print(v)
```

- Producirá la siguiente salida: Contenido del arreglo: [4, 7, 2, 8, 4, 3]

- Si se necesita crear un arreglo de tamaño conocido, y preasignado con valores iguales (por ejemplo ceros) se puede hacer usando el operador * (multiplicación). Ejemplo: crear un arreglo con “n” elementos (n = 4)

```
# Definición arreglo notas con 4 componentes.
```

```
# Cada componente se inicializa con cero.
```

```
notas = 4 * [0]
```

```
print(' El arreglo notas: ', notas)
```

```
#Visualiza por consola [0, 0, 0, 0]
```

- Creado el arreglo, se usa la referencia que lo apunta para acceder a sus componentes, ya sea para consultarlos o modificarlos, colocando a la derecha un par de corchetes y el índice del casillero que se desea acceder

Operaciones posibles con una variable de tipo list.

```

v = [4, 10, 2, 8, 4, 3]
# 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

```

Carga de un arreglo unidimensional

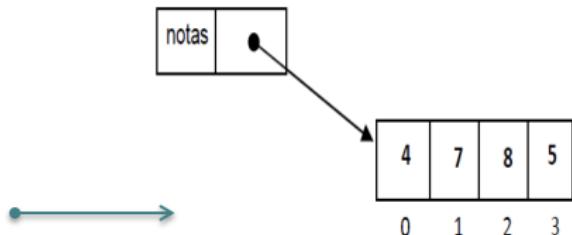
- Incorporación de elementos: si el arreglo se creó vacío se puede agregar elementos por medio de la función `append()`.

```

notas = []
notas.append(4)
notas.append(7)
notas.append(8)
notas.append(5)
print('Notas:', notas)

```

Figura 3: Efecto de la función `append()` al crear una variable de tipo *list*.



La función `append()` agrega el contenido de la variable que es tomado como parámetro a una lista, pero lo hace de forma que ese valor se agrega al final del arreglo

- En el caso de que el arreglo se cree con un número fijo de componentes, se puede usar un ciclo `for` y se usa el índice para acceder a cada casilla y proceder a la carga de elementos.
- El siguiente esquema muestra la forma de hacer la carga de cuatro valores por teclado en el arreglo.

```
notas1 = 4 * [0] # Vector con 4 componentes.
```

```
for i in range(4):
```

```
    # Carga las notas por teclado
```

```
    notas1[i] = int(input('Cargue nota:'))
```

- Otra alternativa que logra el mismo efecto del script anterior, es usar la función len() para conocer el tamaño o cantidad de elementos que tiene una variable de tipo list.

```
notas1 = 4 * [0] # Vector con 4 componentes.
```

```
for i in range(len(notas1)):  
    # Carga las notas por teclado  
    notas1[i] = int(input('Cargue nota:'))
```

Procesamiento secuencial de un arreglo

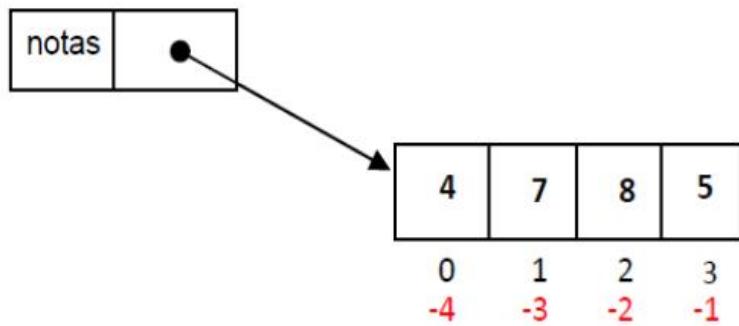
- Si el arreglo está ya creado, se puede procesar a cada uno de los componentes iterando sobre ellos, o sobre sus índices.
- Para ello se utiliza un ciclo for lo que permite un recorrido completo del mismo. Ejemplo: Obtener el promedio de notas del arreglo unidimensional.

```
suma = 0  
notas1 = 4 * [0]  
n = len(notas1)  
for i in range(n):  
    suma += notas1[i]  
promedio = suma / n  
print('El promedio es:', promedio)
```

Índices negativos en un arreglo unidimensional

- En Python una variable de tipo list es dinámica, es decir que se pueden agregar, quitar elementos en cualquier posición o incluso cambiar el tipo de sus componentes.
- Los elementos individuales de una variable de tipo list pueden accederse mediante índices.
- En Python se pueden usar índices negativos con cualquier tipo de secuencia (listas, tuplas, cadenas). Considerando que el índice -1 representa el último elemento, el -2 el penúltimo elemento y así sucesivamente.

Figura 4: Esquema de índices para una secuencia de cualquier tipo en Python.



- Creación de un arreglo unidimensional y diferentes posibilidades de recorridos de izquierda a derecha y viceversa (uso de índices negativos).

```
# crear una lista con n números del 1 al n...
n = 10
numeros = []
for i in range(1, n+1):
    numeros.append(i)
print("Lista original: ", numeros)

# recorre de izquierda a derecha y mostrar los #
# números pares...
print("Valores pares contenidos en la lista: ")
for i in range(len(numeros)):
    if numeros[i] % 2 == 0:
        print(numeros[i])
```

- Recorrido del arreglo unidimensional de derecha a izquierda.

```
# (Continuación del script anterior)

# recorrer de derecha a izquierda y mostrar

# todos los números (uso de índices negativos)...

print("\nContenido de la lista, en forma invertida: ")

for i in range(-1, (-len(numeros) - 1), -1):

    print(numeros[i])
```

Modificación y eliminación de elementos en un arreglo unidimensional

- En Python una variable de tipo list son de naturaleza **mutable**. El siguiente ejemplo cambia el valor del elemento en la posición 2 por el valor -1.

```
numeros = [3, 5, 8, 6]
numeros[2] = -1
print(numeros)
# muestra: [3, 5, -1, 6]
```

- Si se desea eliminar un elemento particular, se puede recurrir la instrucción `del`.

```
numeros = [3, 5, 8, 6, 9, 2]
del numeros[3]
print(numeros) # muestra: [3, 5, 8, 9, 2]
```

Subrangos o Corte de índices

- Todo tipo de secuencia en Python permite acceder a un subrango de casilleros “rebanando” o “cortando” sus índices.

```
numeros = [4, 2, 3, 4, 7]
nueva = numeros[1:4]
print("\nNueva lista: ", nueva)
# muestra: [2, 3, 4]
```

- La siguiente instrucción también usa un corte de índices pero esta vez para copiar una lista.

```
numeros = [2, 4, 6, 5]
copia = numeros[:] # copia toda la lista
print(copia) # muestra: [2, 4, 6, 5]
```

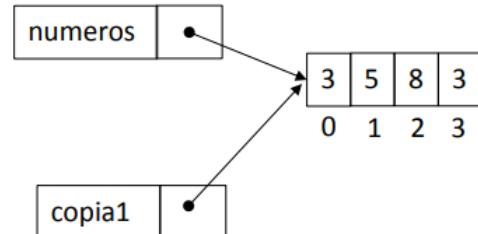
- Notar la diferencia entre asignar entre sí dos variables de tipo list o usar corte de índices para copiar un arreglo completo.

```

numeros = [3, 5, 8, 3]
# una copia a nivel de referencias (copia
# superficial)...
copia1 = numeros # ambas variables apuntan a la
# misma lista...
numeros[2] = -1
print("Original: ", numeros)
print("Copia 1: ", copia1)

numeros: [3, 5, -1, 3]
copia1: [3, 5, -1, 3]

```



- En el caso de usar corte de índices se esta recuperando una copia de elementos. El siguiente script no copia las referencias, sino que crea un segundo arreglo cuyos elementos son iguales a los del primero.

```

numeros = [3, 5, 8, 3] → numeros | • → 3 5 8 3
# una copia a nivel valores...
copia2 = numeros[:] → copia2 | • → 3 5 8 3
numeros[2] = -1
print("Original: ", numeros) → numeros | • → 3 5 -1 3
print("Copia 2: ", copia2) → copia2 | • → 3 5 8 3

numeros: [3, 5, -1, 3]
copia2: [3, 5, 8, 3]

```

Se produce la siguiente salida.

CUESTIONARIO

¿Cuál de las siguientes afirmaciones es CIERTA respecto del uso de índices en un arreglo (representado con una variable de tipo *list*) en Python? Más de una respuesta puede ser válida. Marque todas las que considere correctas.

Seleccione una o más de una:

a.

En Python, el programador puede hacer que los índices de un arreglo comiencen desde un número mayor a cero.

b.

Los índices de un arreglo deben ser numéricos, y valen como índices tanto valores negativos, como cero o positivos.

¡Correcto!



c.

Los índices de un arreglo van numerados de 0 hasta el tamaño del arreglo menos uno.

¡Correcto!



d.

Se pueden usar índices de tipo *cadena de caracteres* para entrar a una casilla de un arreglo

¿Qué hace el siguiente script en Python?

```
n = 20  
v = n * [0]  
for i in range(n):  
    v[i] = 3*i
```

Seleccione una:



a.

Crea un arreglo unidimensional de 20 componentes, y llena ese arreglo con el número 3 (veinte veces el 3).



b.

Crea un arreglo unidimensional de 20 componentes, y llena ese arreglo con los números del 0 al 19.



c.

Crea un arreglo unidimensional de 20 componentes, y llena ese arreglo con 20 números aleatorios.



d.

Crea un arreglo unidimensional de 20 componentes, y llena ese arreglo con los primeros 20 múltiplos de 3.

¡Correcto!

¿Cuál de las siguientes afirmaciones es CIERTA respecto del uso de arreglos (representados con variables de tipo *list*) en Python? Más de una respuesta puede ser válida. Marque todas las que considere correctas.

Seleccione una o más de una:



a.

Siempre se puede saber el tamaño de un arreglo llamando a la función *len()* provista por Python.

¡Correcto!



b.

El primer índice de cada dimensión de un arreglo en Python es siempre cero (a menos que se usen índices negativos).

¡Correcto!



c.

Los arreglos implementados como variables de tipo *list* en Python pueden aumentar o disminuir su tamaño a medida que el programador lo requiera.

¡Correcto!



d.

Un arreglo implementado como una variable de tipo *list* es completamente equivalente a una *tuple* en Python.

Analice el siguiente script en Python:

```
n = 6  
v = n * [0]  
for i in range(n):  
    v[i] = '123'
```

¿Es correcto este script, o existe algún problema con él?

Seleccione una:



a.

Lanza un error de índice fuera de rango y se interrumpe, pues se intenta acceder a una casilla con índice fuera de rango en el ciclo for.



b.

No hay nada de malo con ese segmento.

¡Correcto! Efectivamente, el arreglo comienza con seis ceros, y luego se cambian esos seis ceros por la cadena '123' repetida seis veces... Y esto es válido por ser Python un lenguaje de tipado dinámico.



c.

El arreglo v está mal definido: deben usarse paréntesis (o sea: $v = n * (0)$) en lugar de corchetes (o sea, en lugar de: $v = n * [0]$)



d.

El arreglo fue creado como un arreglo de n valores de tipo *int*, y por lo tanto no pueden asignarse luego cadenas de caracteres en sus casilleros.

Analice el siguiente script en Python:

```
n = 6
v = n * [0]
for i in range(n+1):
    v[i] = i
```

¿Hay algún problema con el script mostrado?

Seleccione una:



a.

Lanza un error y se interrumpe, pues en el ciclo *for* se intenta acceder a una casilla no definida.

¡Correcto! En efecto, el ciclo *for* está incluyendo una vuelta con $i = 6\dots$ y el índice de la última casilla es 5...



b.

El arreglo está mal definido: la instrucción $v = n * [0]$ está creando un arreglo vacío, sin casilleros.



c.

Convierte cada casilla del vector a una cadena de caracteres.



d.

No hay nada de malo con ese script.

Analice el siguiente script en Python:

```
v = [2, 4, 1, 6]
v[0] = v[v[0]] * 3
print('v[0]:', v[0])
```

¿Cuál de las siguientes es correcta en relación al script mostrado?

Seleccione una:



a.

Se mostrará el mensaje: $v[0] = 2$.



b.

Se mostrará el mensaje: $v[0] = 12$.



c.

Se mostrará el mensaje: $v[0] = 3$.

¡Correcto!



d.

Se lanzará un error y el script se interrumpirá.

Analice el siguiente script en Python:

```
n = 5  
a = n * [0]  
for i in range(n):  
    a[i] = i + 1  
  
v = n * [0]  
for i in range(n):  
    v[a[i]] = a[i]
```

¿Cuál de las siguientes es correcta en relación al script mostrado?

Seleccione una:



a.

Todos los casilleros del arreglo a se convierten a *float* y se vuelven a asignar en el mismo arreglo a .



b.

El script lanza un error y se interrumpe por intentar acceder a una casilla fuera de rango en el arreglo v .

¡Correcto! Los valores de los casilleros de a se están usando como índices para entrar en v ... pero el último casillero de a está valiendo 5, con lo cual se produce un error en la última vuelta del segundo *for*.



c.

El arreglo v queda valiendo los mismos valores que el arreglo a , pero convertidos al tipo *float*.



d.

Tanto el arreglo a como el v quedan con todos sus casilleros valiendo 0.

¿Qué hace el siguiente programa en Python?

```

__author__ = 'Cátedra de AED'

def test():
    n = 10
    v = n * [0]

    for i in range(n):
        v[i] = int(input('v[' + str(i) + ']: '))

    im = 0
    for i in range(1, n):
        if v[i] < v[im]:
            im = i

    print('El valor pedido es:', v[im])

if __name__ == '__main__':
    test()

```

Seleccione una:



a.

Carga por teclado un arreglo unidimensional con 10 números enteros. Luego busca el menor valor contenido en el arreglo y muestra ese menor.

JCorrecto!



b.

Carga por teclado un arreglo unidimensional con 10 números enteros. Luego busca el mayor valor contenido en el arreglo y muestra ese mayor.



c.

Carga por teclado un arreglo con 10 números enteros y luego ordena y muestra ese arreglo.



d.

Carga por teclado un arreglo unidimensional con 10 números enteros. Luego calcula y muestra el promedio de los elementos contenidos en ese arreglo.

¿Qué hace la siguiente función en Python?

```

def comprobar(v):
    n = len(v)
    for i in range(n-1):
        if v[i] > v[i+1]:
            return False
    return True

```

Seleccione una:



a.

Retorna *True* si el arreglo *v* tomado como parámetro contiene al valor *n*, o retorna *False* en caso contrario.



b.

Retorna *True* si el arreglo *v* tomado como parámetro contiene todos sus elementos iguales, o retorna *False* en caso contrario.



c.

Retorna *True* si el arreglo *v* tomado como parámetro está ordenado de menor a mayor, o retorna *False* en caso contrario.



d.

Retorna *True* si el arreglo *v* tomado como parámetro está ordenado de mayor a menor, o retorna *False* en caso contrario.

¿Qué hace la siguiente función en Python?

```
def generar(v):
    n = len(v)

    ac = 0
    for i in range(n):
        ac += v[i]
    p = ac / n

    c = 0
    for i in range(n):
        if v[i] >= p:
            c += 1

    mp = c * [0]
    idx = 0
    for i in range(n):
        if v[i] >= p:
            mp[idx] = v[i]
            idx += 1

    return mp
```

Seleccione una:



a.

Toma un arreglo *v* como parámetro. Calcula el promedio de los valores contenidos en *v*. Finalmente, genera un segundo arreglo *mp* que contiene sólo los elementos de *v* que son mayores o iguales al promedio y retorna el nuevo arreglo *mp*.

¡Correcto!

b.

Toma un arreglo `v` como parámetro. Genera un segundo arreglo `mp` que contiene solo los elementos no negativos de `v`, y retorna el nuevo arreglo `mp`.

c.

Toma un arreglo `v` como parámetro. Genera un segundo arreglo `mp` que contiene sólo los elementos de `v` que son mayores a 10 y retorna el nuevo arreglo `mp`.

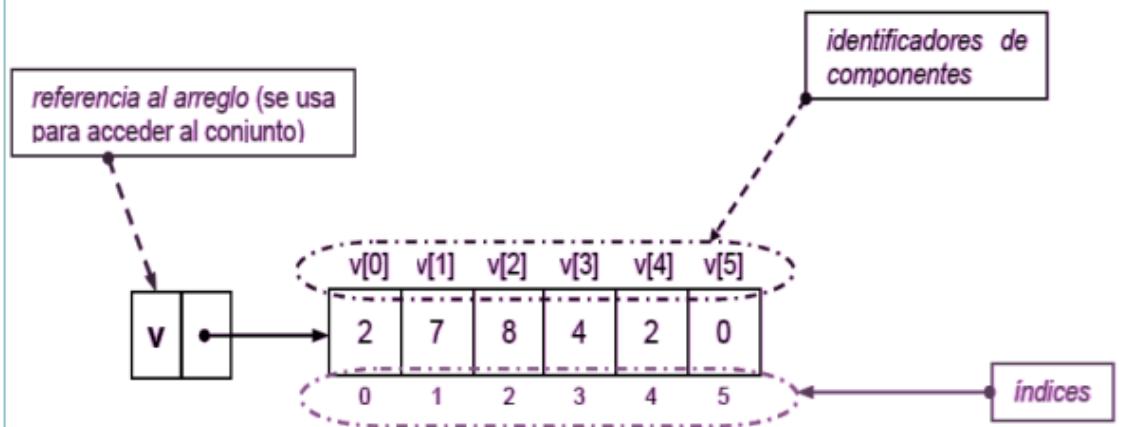
d.

Toma un arreglo `v` como parámetro. Ordena el arreglo `v`. Finalmente, genera un segundo arreglo `mp` que contiene todos los elementos de `v`, y retorna `mp` (que entonces, será igual al vector `v` pero ordenado).

FICHA 13 Arreglos. Algoritmos y Técnicas Básicas

- Arreglo unidimensional: es una colección de valores que se organiza de tal forma, que cada componente individual es identificado por un número denominado índice.
- Python provee un tipo de secuencia que permite representar a un arreglo unidimensional, denominada `list` o lista. Se caracteriza por ser mutable

Figura 1: Esquema básico de un arreglo unidimensional.



- La creación del arreglo unidimensional puede realizarse de diferentes maneras: i) Creando un arreglo vacío. Por ejemplo:

`notas = []` o `notas = list()`

- ii) En el caso de conocer previamente los valores a almacenar, se puede crearlo enumerando sus componentes:

`notas = [9, 10, 7, 6, 7]`

iii) Estableciendo la cantidad de componentes del arreglo, y que almacene el mismo valor en cada casilla. Para esto se usa el operador *.

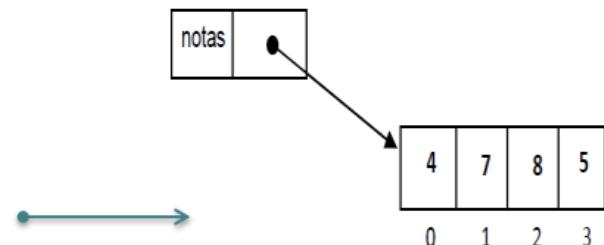
```
notas = 4 * [0]
```

- Si el arreglo se crea vacío, la carga se puede realizar por medio del método append().

Ejemplo:

```
notas = []
notas.append(4)
notas.append(7)
notas.append(8)
notas.append(5)
print('Notas:', notas)
```

Figura 3: Efecto de la función `append()` al crear una variable de tipo `list`.



La función append() agrega el contenido de la variable que es tomado como parámetro a una lista, pero lo hace de forma que ese valor se agrega al final del arreglo

- En el caso que el arreglo no se cree vacío, se puede agregar elementos solicitando al usuario que cargue los datos por teclado.
- En este caso para procesar un arreglo es común el uso del ciclo for.
- El siguiente script muestra la forma de hacer la carga en esta situación.

```
notas1 = 4 * [0] # Arreglo con 4 componentes.
for i in range(4):
    # Carga las notas por teclado
    notas1[i]= int(input('Cargue nota: '))
print('La carga del arreglo ha finalizado!')
```

Arreglos: Visualización de elementos

- Para visualizar los elementos almacenados en un arreglo se puede utilizar en forma directa la instrucción print().
- Si la visualización de los elementos contenidos en un arreglo se quiere hacer de una manera más personalizada, en estos casos se recurre a la utilización de un ciclo for. Ejemplo:

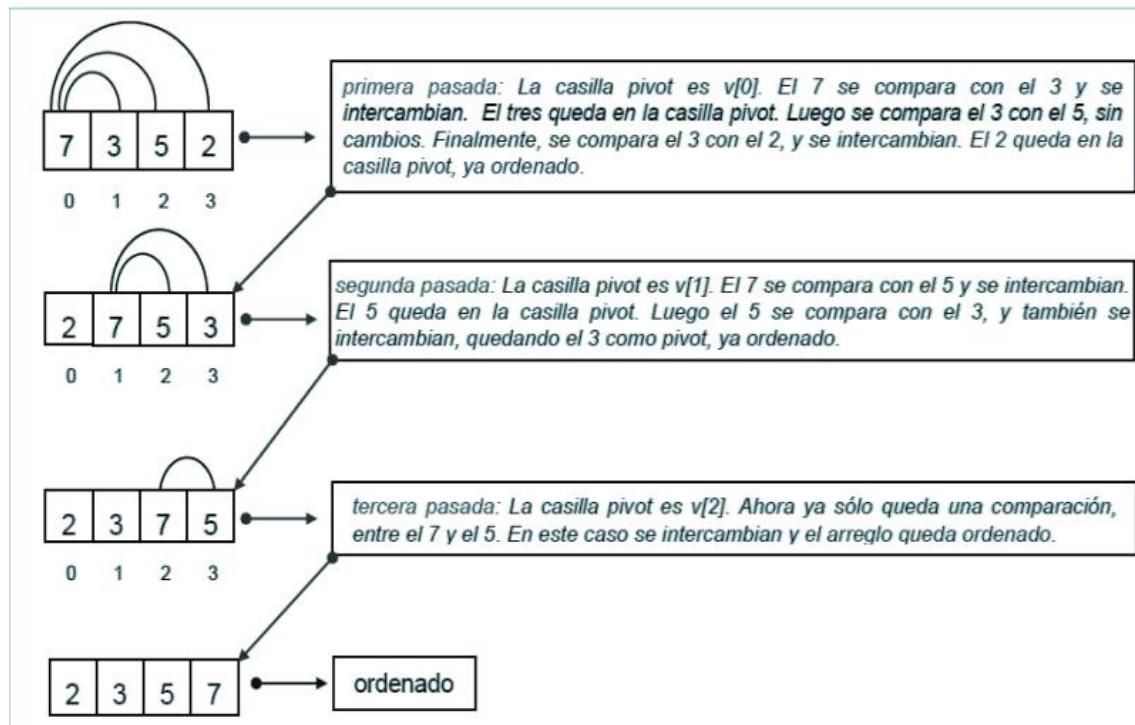
```
# Tomando como base el arreglo con 4 componentes anterior.
```

```
for i in range(4):
    # Visualización de elementos.
    print('En la posición:', str(i), 'la nota es:',notas1[i])
```

Ordenamiento de arreglo unidimensional

- Una operación muy común cuando se trabaja con arreglos unidimensionales es el de ordenar sus elementos ya sea en orden ascendente o descendente.
- La ventaja del ordenamiento en cualquier estructura de datos radica fundamentalmente en la operación de búsqueda de un elemento. Ya que si la estructura tiene sus elementos ordenados la búsqueda es más rápida.
- Para realizar esta tarea existen diversos algoritmos, uno de ellos es el que se denomina Ordenamiento de Selección Directa.

Algoritmo de Ordenamiento: Selección Directa



```
def selection_sort(v):  
    # ordenamiento por selección 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]
```

```

__author__ = 'Cátedra de AED'

def principal():
    v = [3, 2, 7, 1]
    # Ordenar el arreglo.
    selection_sort(v)
    print('\nEl vector ordenado es:', v)

if __name__=='__main__':
    principal()

#La visualización por consola es
[1,2,3,7]

```

Búsqueda Secuencial

- Este tipo de búsqueda permite resolver el problema de conocer si un determinado valor “x” se encuentra contenido o no un arreglo de “n” componentes.
- En esta búsqueda la mecánica a seguir es: si se encuentra el elemento se informa la posición y si no se encuentra se informa con un mensaje tal situación.
- Este tipo de búsqueda se puede aplicar cuando el arreglo esta desordenado o no se conoce nada respecto a su estado.

```

# arreglos.py (módulo)

def linear_search(v, x):
    # búsqueda secuencial...
    for i in range(len(v)):
        if x == v[i]:
            return i
    return -1

```

```
__author__ = 'Cátedra de AED'

import arreglos

def principal():
    v =[ 3, 2, 7, 1]
    x = int(input('Valor a buscar en el arreglo: '))
    # aplicar búsqueda secuencial...
    ind = arreglos.linear_search(v, x)
    # notificar el resultado de la búsqueda...
    if ind >= 0:
        print('\nEstá en la casilla', ind)
    else:
        print('\nNo está en el arreglo')

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

- Este tipo de búsqueda es mucho mas eficiente que la búsqueda secuencial y solo puede ser aplicada si el arreglo “v” esta ordenado.

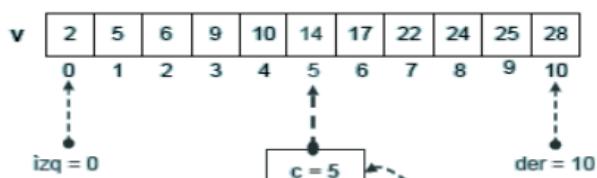
- La idea básica del algoritmo es:

- | Uso de dos índices auxiliares (izq y der).
- | En el intervalo marcado por izq y der se toma el elemento central cuyo índice es c:
 $c = (izq + der) // 2$
- | Se verifica si el valor contenido en v [c] coincide con el número buscado. Si coincide termina la búsqueda. En caso contrario se aprovecha el ordenamiento del arreglo.

n = 11

valor a buscar: $x = 10$

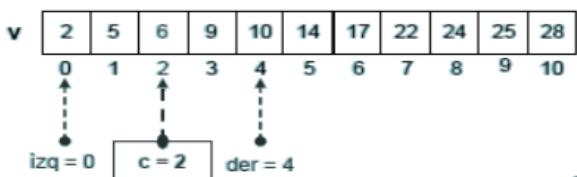
Paso 1



n = 11

valor a buscar: $x = 10$

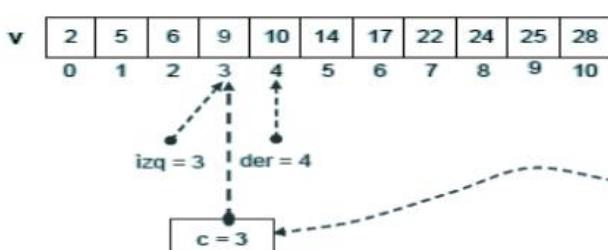
Paso 2



n = 11

valor a buscar: $x = 10$

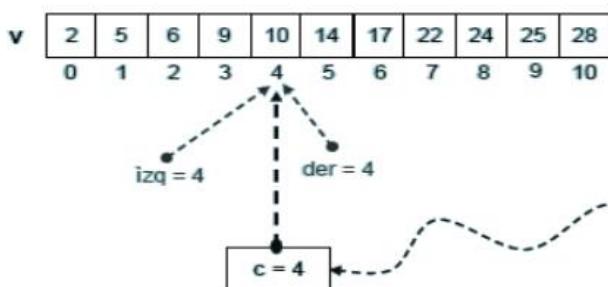
Paso 3



n = 11

valor a buscar: $x = 10$

Paso 4



```
# arreglos.py (módulo)

def binary_search(v, x):
    # búsqueda binaria...
    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
```

```
__author__ = 'Cátedra de AED'

import arreglos

def principal():
    v = [1, 2, 3, 7]
    x = int(input('Valor a buscar en el arreglo: '))
    # aplicar búsqueda binaria...
    ind = arreglos.binary_search(v, x)
    # notificar el resultado de la búsqueda...
    if ind >= 0:
        print('\nEstá en la casilla', ind)
    else:
        print('\nNo está en el arreglo')

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

CUESTIONARIOS

¿Cuál es la **idea principal** en la que se basa el algoritmo de ordenamiento de **Selección Directa** que se presentó en clase? (Suponga que se desea ordenar el arreglo de menor a mayor).

Seleccione una:



a.

Realizar varias pasadas sobre el arreglo, en cada pasada **seleccionar el menor valor** de los que aún no han sido ordenados, y colocarlo en su casilla definitiva.

¡Correcto!



b.

Realizar varias pasadas sobre el arreglo, en cada pasada **comparar a cada elemento con el que le sigue inmediatamente**, intercambiarlos si están desordenados y seguir haciendo pasadas en esta forma hasta que el arreglo quede ordenado.



c.

Realizar varias pasadas sobre el arreglo. En cada pasada seleccionar un elemento cualquiera x . Pasar a la derecha del arreglo todos los valores mayores a x . Pasar a la izquierda del arreglo todos los valores menores a x . Finalmente, aplicar la misma idea a cada una de las dos "mitades" así obtenidas y seguir así hasta que ya no pueda seguir obteniendo nuevas "mitades".



d.

Realizar **una sola pasada sobre el arreglo**, en esa pasada comparar a cada elemento con el que le sigue e intercambiarlos si están invertidos.

La siguiente función implementa el algoritmo de ordenamiento por *Selección Directa*, tal como se analizó en clases para ordenar de menor a mayor un arreglo v con n componentes:

```
def selection_sort(v):
    # ordenamiento por selección 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]
```

¿Qué efecto se produciría en la función anterior si la instrucción condicional `if v[i] > v[j]:` fuese reemplazada por `if v[i] < v[j]:`?

Seleccione una:



a.

No causaría ningún efecto particular: el arreglo seguiría siendo ordenado de menor a mayor.



b.

Provocaría que el arreglo permanezca siempre sin cambio alguno (no será nunca ordenado de forma alguna, ni se modificará nunca su contenido original).



c.

Provocará que el programa se interrumpa con un mensaje de error en la primera comparación.



d.

El arreglo sería ordenado, pero ahora de mayor a menor.

¡Correcto!

¿Cuántas comparaciones hace el algoritmo de Búsqueda Secuencial para encontrar un valor x en un arreglo de n componentes en el *peor caso posible*? (El peor caso es el que obliga a un algoritmo a hacer la máxima cantidad de trabajo. Obviamente, en el caso de la Búsqueda Secuencial ese peor caso se presenta si el valor buscado está exactamente en la última casilla, o bien, si el valr buscado no está en el arreglo).

Seleccione una:



a.

$\log_2(n)$ comparaciones.



b.

Una sola comparación, siempre.



c.

n comparaciones.

¡Correcto!



d.

n^2 comparaciones.

Oportunamente se presentó en clases el algoritmo de *búsqueda secuencial*, el cual toma un arreglo, busca un valor x en el mismo casilla por casilla, y retorna el índice de la casilla que lo contiene (si x está en el arreglo) o retorna -1 si x no está en el arreglo. Suponga que propone la siguiente variante para el algoritmo de búsqueda secuencial:

```
def linear_search(v, x):
    r = -1
    for i in range(len(v)):
        if x == v[i]:
            r = i
```

```
        return r
```

¿Hay algún inconveniente o problema con esta variante?

Seleccione una:



a.

Hay un inconveniente: la variante propuesta encuentra el valor buscado (si existía) pero no corta el ciclo de búsqueda al encontrarlo con lo que siempre se hacen n comparaciones.

¡Correcto!



b.

La variante propuesta funciona solamente si el valor x está en el arreglo (falla si x no está).



c.

La variante propuesta no funciona correctamente: sólo llega a analizar el contenido de la primera casilla.



d.

La variante propuesta funciona correctamente.

Oportunamente se presentó en clases el algoritmo de **búsqueda secuencial**, el cual toma un arreglo, busca un valor x en el mismo casilla por casilla, y retorna el índice de la casilla que lo contiene (si x está en el arreglo) o retorna -1 si x no está en el arreglo. Suponga que propone la siguiente variante para el algoritmo de búsqueda secuencial:

```
def linear_search(v, x):
    n = len(v)
    for i in range(n):
        if x == v[i]:
            return i
        else:
            return -1
```

¿Funciona correctamente esta variante? Si no funciona, ¿cuál es el problema?

Seleccione una:



a.

La variante propuesta provoca un error de intérprete: un **if** no puede tener una instrucción **return** en cada una de sus ramas.



b.

La variante propuesta funciona solamente si el valor x está en el arreglo (falla si x no está).



c.

La variante propuesta no funciona correctamente: sólo llega a analizar el contenido de la primera casilla.

¡Correcto!



d.

La variante propuesta funciona correctamente.

¿En cuáles de los siguientes casos es aplicable el algoritmo de **búsqueda binaria** en un arreglo? (Seleccione todas las respuestas que considere correctas)

Seleccione una o más de una:



a.

El arreglo en el cual se debe realizar la búsqueda está ordenado y permanecerá sin cambios.

¡Correcto!



b.

El arreglo en el cual se debe realizar la búsqueda está desordenado, se nos permite ordenarlo, luego permanecerá sin cambios y debemos realizar muchas búsquedas.

¡Correcto!



c.

El arreglo en el cual se debe realizar la búsqueda está desordenado, se nos permite ordenarlo, luego podrá alterarse el contenido (quedando eventualmente desordenado) y debemos realizar varias búsquedas.



d.

El arreglo en el cual se debe realizar la búsqueda está desordenado y no se nos permite ordenarlo.

Oportunamente se presentó en clases el algoritmo de **búsqueda binaria**, el cual toma un arreglo ordenado, busca un valor x en el mismo, y retorna el índice de la casilla que lo contiene (si x está en el arreglo) o retorna -1 si x no está en el arreglo. Suponga que propone la siguiente variante para el algoritmo de búsqueda binaria:

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

```
        return -1
```

```
    if x < v[c]:  
        der = c - 1  
    else:  
        izq = c + 1
```

```
    return -1
```

¿Funciona correctamente esta variante? Si no funciona, ¿cuál es el problema?

Seleccione una:



a.

La variante propuesta provoca un error de intérprete y no llega a arrancar: el segundo `if` incluido dentro del ciclo nunca puede ejecutarse, ya que las dos ramas del `if` anterior terminan con una instrucción `return`.



b.

La variante propuesta no funciona correctamente: sólo llega a analizar el contenido de la casilla del centro del arreglo.

¡Correcto!



c.

La variante propuesta funciona solamente si el valor `x` está en el arreglo (falla si `x` no está).



d.

La variante propuesta funciona correctamente.

Oportunamente se presentó en clases el algoritmo de **búsqueda binaria**, el cual toma un arreglo ordenado, busca un valor `x` en el mismo, y retorna el índice de la casilla que lo contiene (si `x` está en el arreglo) o retorna `-1` si `x` no está en el arreglo. Mostramos el algoritmo para dar mejor contexto a la pregunta:

```
def binary_search(v, x):  
    # búsqueda 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
```

¿Qué pasaría con la función anterior si el arreglo `v` contuviese *cadenas de caracteres* en cada casillero (en lugar de números), y el parámetro `x` contuviese también una cadena de caracteres?

Seleccione una:



a.

La función mostrada provocaría un error en tiempo de ejecución y se interrumpiría el programa al intentar determinar si x menor o mayor que $v[i]$.



b.

La función mostrada funcionaría correctamente de todos modos.

¡Correcto!



c.

La función mostrada no provocaría que el programa se interrumpa, pero funcionaría en forma incorrecta y retornaría siempre -1 (nunca encontraría la cadena x buscada).



d.

La función mostrada funciona solamente si el valor x está en el arreglo (falla si x no está, interrumpiendo el programa con un mensaje de error).

¿Qué pasaría con el algoritmo de *Búsqueda Binaria* si el valor buscado x estuviese repetido más de una vez en el arreglo?

Seleccione una:



a.

El algoritmo produciría un error en tiempo de ejecución y se interrumpiría el programa.



b.

El algoritmo no provocaría que el programa se interrumpa, pero funcionaría en forma incorrecta y retornaría siempre -1 (nunca encontraría *el valor* x buscado).



c.

El algoritmo funciona solamente si el valor x está en el arreglo (falla si x no está, interrumpiendo el programa con un mensaje de error).



d.

El algoritmo funcionaría correctamente de todos modos: retornaría el índice de la primera casilla encontrada que contenga a x .

¡Correcto!

¿En cuáles de los siguientes casos es aplicable el algoritmo de *Fusión de Arreglos* que se describió en las fichas de clases para producir un tercer arreglo ordenado? (Seleccione todas las respuestas que considere correctas)

Seleccione una o más de una:

a.

El proceso es aplicable sin importar si los dos arreglos originales están ordenados o no, ya que el algoritmo de fusión analizado primero ordena esos dos arreglos, y luego procede a la fusión.

b.

Uno de los arreglos originales debe estar ordenado y el otro arreglo puede estar desordenado.

c.

Los arreglos originales deben estar desordenados.

d.

Los arreglos originales deben estar ordenados de menor a mayor si se quiere producir un tercer arreglo ordenado de menor a mayor.

¡Correcto!

FICHA 14 Arreglos: Caso de Estudio I

Arreglos correspondientes o **paralelos**

- Los arreglos “correspondientes” o “paralelos” se utilizan cuando es necesario almacenar información en varios arreglos unidimensionales a la vez. De tal forma que exista una correspondencia entre los valores almacenados en las casillas con el mismo índice (llamadas casillas homólogas).
- Por ejemplo: puede ser que sea necesario guardar en un arreglo los nombres de ciertas personas y en otro el importe del sueldo que perciben

<i>nombres</i>	Juan	Ana	Alejandro	María	Pedro
<i>sueldos</i>	1100	2100	1300	750	800
	0	1	2	3	4

↑

Ambos arreglos deben tener la misma cantidad de componentes

Acceso y visualización:

- El manejo de ambos arreglos es simple: solo se debe recordar que hay que usar el mismo índice en ambos arreglos para acceder a la información de la misma persona. Siguiendo este esquema, visualiza por pantalla el nombre y el sueldo de la persona en la casilla 2.
- Ejemplo: asumiendo que los arreglos ya están cargados: print('Nombre:', nombres[2]) # Alejandro print('Sueldo:', sueldos[2]) #1300

<i>nombres</i>	Juan	Ana	Alejandro	María	Pedro
<i>sueldos</i>	1100	2100	1300	750	800
	0	1	2	3	4

Carga y visualización

- El primer script permite la carga de los dos arreglos paralelos. El primer arreglo contiene el nombre, mientras que en el segundo arreglo almacena el sueldo que percibe el empleado.

```
def read(nombres, sueldos):
  n = len(nombres)
  for i in range(n):
    nombres[i] = input('Ingrese nombre: ')
    sueldos[i] = int(input('Ingrese sueldo: '))
```

- El segundo script visualiza los datos contenidos en los arreglos.

```
def display(nombres, sueldos):
  n = len(nombres)
  print('Datos de los empleados:')
  for i in range(n):
    print('Nombre:', nombres[i], '- Sueldo:', sueldos[i])
```

Script completo: arreglos paralelos

```

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cant. elementos (mayor a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Se pidió mayor a', inf, '... cargue de nuevo...')
    return n

def test():
    n = validate(0)
    nombres = n * [' ']
    sueldos = n * [0]
    print('\nCargue los datos de las personas:')
    read(nombres, sueldos)
    display(nombres, sueldos)

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

```

- Desarrollar un programa que permita cargar tres arreglos con n nombres de personas, sus edades y los sueldos que ganan. Luego de realizar la carga de todos los datos. Se solicita:
 - Informar el nombre de todos los empleados mayores a 18 años que cobren un sueldo mayor a 10.000 ordenados alfabéticamente

<i>nombres</i>	Juan	Ana	Alejandro	Maria	Pedro
<i>edades</i>	30	18	25	43	38
<i>sueldos</i>	8000	15000	9000	40000	16000
	0	1	2	3	4

- Para resolver este caso de estudio necesitamos implementar tres funciones: i) una que cargue los arreglos, ii) otra que ordene los elementos del arreglo, iii) y otra que muestre el listado solicitado.
- El script siguiente muestra la implementación de la primera función.

```

def read(nombres, edades, sueldos):
    n = len(nombres)
    for i in range(n):
        nombres[i] = input('Nombre[' + str(i) + ']: ')
        edades[i] = int(input('Edad: '))
        sueldos[i] = int(input('Sueldo: '))
    print()

```



nombres	Juan	Ana	Alejandro	María	Pedro
edades	30	18	25	43	38
sueldos	8000	15000	9000	40000	16000

0 1 2 3 4

- A continuación se muestran las funciones: ii) que ordena los elementos del arreglo (Selección directa), iii) y otra que muestre el listado solicitado

```

def sort(nombres, edades, sueldos):
    n = len(nombres)
    for i in range(n-1):
        for j in range(i+1, n):
            if nombres[i] > nombres[j]:
                nombres[i], nombres[j] = nombres[j], nombres[i]
                edades[i], edades[j] = edades[j], edades[i]
                sueldos[i], sueldos[j] = sueldos[j], sueldos[i]

```

```

def display(nombres, edades, sueldos):
    n = len(nombres)
    print('Mayores de 18 que ganan menos de 10000 pesos:')
    for i in range(n):
        if edades[i] > 18 and sueldos[i] < 10000:
            print('Nombre:', nombres[i], '- Edad:', edades[i], '- Sueldo:', sueldos[i])

```

- El siguiente es el script principal que activa las funciones anteriormente implementadas.

```

def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cant. de elementos (mayor a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidió mayor a', inf, '... cargue de nuevo...')
    return n

def test():
    n = validate(0)
    nombres = n * [' ']
    edades = n * [0]
    sueldos = n * [0]
    print('\nCargue los datos de las personas:')
    read(nombres, edades, sueldos)
    sort(nombres, edades, sueldos)
    display(nombres, edades, sueldos)

if __name__ == '__main__':
    test()

```

Vectores de conteo y de acumulación

- Los arreglos son estructuras de datos muy útiles, porque facilitan el acceso directo a un componente.
- En algunos casos esta característica facilita la resolución de problemas de manera simple y rápida. Por ejemplo: búsqueda binaria.
- Un vector de conteo representa un arreglo en donde cada uno de sus componentes se comporta con un contador.
- Mientras que un vector de acumulación cada componente se comporta como un acumulador.

Forma de vector de conteo

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

Forma de vector de acumulación

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

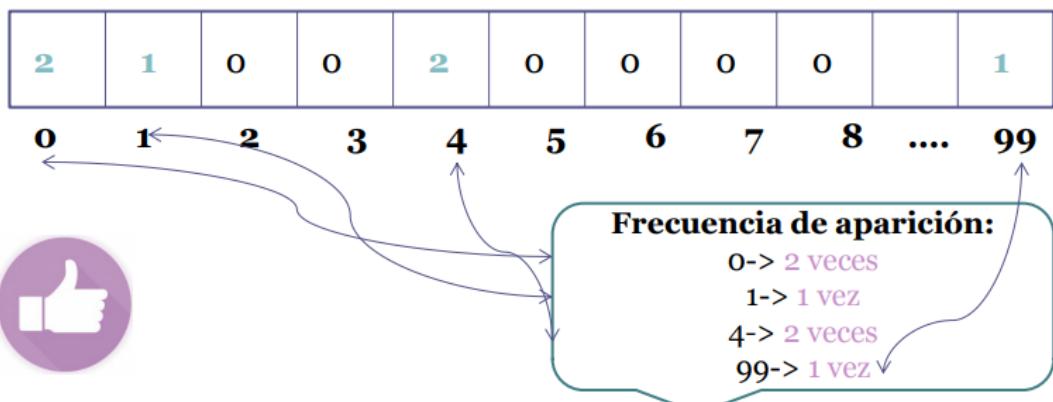
- Cargar por teclado un conjunto de valores tales que todos ellos estén comprendidos entre 0 y 99 (incluidos ambos). Se indica el fin de datos con el número -1. Determinar cuantas veces apareció cada número.

Una estrategia de resolución “eficiente”:

- Usar un arreglo unidimensional “c” de 100 componentes (vector de conteo), de forma que cada componente se use como uno de los contadores que se están necesitando.
- Se coloca inicialmente en cero cada componente del vector (dado que cada componente será un contador).
- Se usa un ciclo para cargar los valores num.
- Considerar el uso de acceso directo del arreglo unidimensional “c” para efectuar el conteo de cada numero ingresado.

Una posible estrategia de resolución:

- La idea central es que si num vale 0 (cero), entonces se debe usar `c[0]` para contararlo, si num vale 1 (uno) usar `c[1]`....
- En general para cada valor de la variable num, tal que: $0 \leq \text{num} \leq 99$, debe usarse `c[num]` para contararlo.
- Ejemplo: Para la secuencia 0, 4, 0, 4, 1, 99, -1



```
def test():
    n = 100
    c = n * [0]

    num = int(input('Ingrese un valor entre 0 y 99 (con -1 corta): '))
    while num != -1:
        if 0 <= num < n:
            c[num] += 1
        else:
            print('Error... el número debía ser >= 0 y <', n)
        num = int(input('Ingrese otro valor entre 0 y 99 (con -1 corta): '))

    print('Resultados:')
    for i in range(n):
        if c[i] != 0:
            print('Número', i, '- Frecuencia de aparición', c[i])

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

CUESTIONARIO

El siguiente programa crea tres arreglos paralelos para almacenar los legajos, los nombres y los promedios de n estudiantes y luego los carga por teclado. ¿Qué muestra finalmente este programa al ejecutarlo?

```
__author__ = 'Cátedra de AED'

def read(leg, nom, pro):
    n = len(leg)
    for i in range(n):
        leg[i] = int(input('Legajo: '))
        nom[i] = input('Nombre: ')
```

```

        pro[i] = float(input('Promedio: '))

def procesar(leg, nom, pro):
    im = 0
    for i in range(1, len(pro)):
        if pro[i] > pro[im]:
            im = i
    return leg[im], nom[im], pro[im]

def test():
    n = int(input('Cantidad de alumnos: '))
    leg = n * [0]
    nom = n * ['']
    pro = n * [0.0]
    read(leg, nom, pro)

    r = procesar(leg, nom, pro)

    print('Datos del estudiante pedido...')
    print('Legajo:', r[0])
    print('Nombre:', r[1])
    print('Promedio:', r[2])

if __name__ == '__main__':
    test()

```

Seleccione una:



a.

Busca el estudiante con mayor promedio del arreglo (mediante la función *procesar()*) y muestra sólo el valor de ese promedio.



b.

Busca el estudiante con mayor promedio del arreglo (mediante la función *procesar()*) y muestra sus datos completos.



c.

Busca el estudiante con mayor promedio del arreglo (mediante la función *procesar()*) y muestra el índice de la casilla del arreglo que lo contiene.



d.

Busca el estudiante con menor promedio del arreglo (mediante la función *procesar()*) y muestra sus datos completos.

El siguiente programa crea y carga un arreglo *numeros* con *n* números enteros y luego procesa ese arreglo con la función *control()* ¿Qué hace exactamente esa función al ejecutarse?

```

__author__ = 'Cátedra de AED'

def read(numeros):
    n = len(numeros)

```

```

        for i in range(n):
            numeros[i] = int(input('Valor[' + str(i) + ']: '))

def control(numeros):
    n = len(numeros)
    for i in range(n-1):
        if numeros[i] != numeros[i+1]:
            return False
    return True

def test():
    n = int(input('Cantidad de números a cargar: '))
    numeros = n * [0]
    read(numeros)

    print()
    if control(numeros):
        print('El contenido del arreglo es correcto')
    else:
        print('El contenido del arreglo no es correcto')

if __name__ == '__main__':
    test()

```

Seleccione una:



a.

Retorna True si todos los números del arreglo son iguales.

¡Correcto!



b.

Retorna True si los números del arreglo no son todos iguales (hay dos o más números diferentes).



c.

Retorna True si el arreglo está ordenado de mayor a menor.



d.

Retorna True si el arreglo está ordenado de menor a mayor.

El siguiente programa crea y carga un arreglo con n números enteros y luego procesa ese arreglo mediante las funciones *conteo()* y *suma()* ¿Qué hacen exactamente esas dos funciones?

```
__author__ = 'Cátedra de AED'
```

```

def read(numeros):
    n = len(numeros)
    for i in range(n):
        numeros[i] = int(input('Valor para la casilla ' + str(i) + ': '))

def conteo(numeros):

```

```

c = 0
for x in numeros:
    if x > 0:
        c += 1
return c

def suma(numeros):
    s = 0
    n = len(numeros)
    for i in range(0, n, 2):
        s += numeros[i]
    return s

def test():
    n = int(input('Cantidad de números a cargar: '))
    numeros = n * [0]
    read(numeros)

    cmc = conteo(numeros)
    scp = suma(numeros)

    print('Conteo pedido:', cmc)
    print('Suma pedida:', scp)

if __name__ == '__main__':
    test()

```

Seleccione una:



a.

La función *conteo()* cuenta los números mayores a 0 almacenados en casillas con índice impar que hay en el arreglo *numeros*, y la función *suma()* acumula los valores contenidos en las casillas con índice par del arreglo *numeros*.



b.

La función *conteo()* cuenta los números mayores a 0 que hay en el arreglo *numeros*, y la función *suma()* acumula los valores contenidos en las casillas con índice par del arreglo *numeros*.

¡Correcto!



c.

La función *conteo()* cuenta los números mayores a 0 que hay en el arreglo *numeros*, y la función *suma()* acumula los valores del arreglo *numeros*.



d.

La función *conteo()* cuenta los números mayores a 0 que hay en el arreglo *numeros*, y la función *suma()* acumula los valores contenidos en las casillas con índice impar del arreglo *numeros*.

El siguiente programa crea y carga un arreglo *ventas* con *n* montos de ventas realizadas por un comercio en distintos momentos. ¿Qué hace exactamente el programa completo al ejecutarse?

```
__author__ = 'Cátedra de AED'

def read(ventas):
    n = len(ventas)
    for i in range(n):
        ventas[i] = int(input('Monto de venta[' + str(i) + ']: '))

def sort(ventas):
    n = len(ventas)
    for i in range(n-1):
        for j in range(i+1, n):
            if ventas[i] < ventas[j]:
                ventas[i], ventas[j] = ventas[j], ventas[i]

def display(ventas, cant):
    n = len(ventas)
    if cant > n:
        print('La cantidad de ventas registradas no alcanza para el listado'
pedido...!')
    return

    print('Montos de las', cant, 'ventas pedidas:')
    for i in range(cant):
        print('Monto[', i, ']:', ventas[i])

def test():
    n = int(input('Cantidad de ventas a cargar: '))
    ventas = n * [0.0]
    read(ventas)

    sort(ventas)

    print()
    display(ventas, 3)

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

Seleccione una:



a.

Muestra siempre todos los montos del arreglo *ventas*, en el mismo orden que tenían en el arreglo *ventas*.



b.

Muestra siempre todos los montos del arreglo *ventas*, ordenados de mayor a menor.



c.

Muestra los *cant* mayores montos del arreglo *ventas*, ordenados de menor a mayor.



d.

Muestra los *cnt* mayores montos del arreglo *ventas*, ordenados de mayor a menor.

JCorrecto!

El siguiente programa crea y carga un arreglo *temp* con *n* valores de temperaturas medidas en diferentes momentos y luego procesa ese arreglo mediante la función *amplitud()* ¿Qué hace exactamente esa función?

```
__author__ = 'Cátedra de AED'

def read(temp):
    n = len(temp)
    for i in range(n):
        temp[i] = int(input('Temperatura[' + str(i) + ']: '))

def amplitud(temp):
    n = len(temp)
    my = mn = temp[0]
    for i in range(1, n):
        if temp[i] > my:
            my = temp[i]
        elif temp[i] < mn:
            mn = temp[i]

    return my - mn

def test():
    n = int(input('Cantidad de temperaturas a cargar: '))
    temp = n * [0.0]
    read(temp)

    d = amplitud(temp)

    print('Amplitud térmica:', d)

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

Seleccione una:



a.

Calcula y retorna la menor temperatura del arreglo *temp*.



b.

Calcula y retorna la la mayor temperatura del arreglo *temp*.



c.

Calcula y retorna la diferencia entre la mayor y la menor temperatura del arreglo *temp*.

JCorrecto!



d.

Calcula y retorna el promedio entre la mayor y la menor temperatura del arreglo *temp*.

El siguiente programa crea tres arreglos para cargar y almacenar en ellos los legajos, los nombres y los promedios de n estudiantes. ¿Qué hace finalmente este programa al ejecutarlo?

```
__author__ = 'Cátedra de AED'

def read(leg, nom, pro):
    n = len(leg)
    for i in range(n):
        leg[i] = int(input('Legajo: '))
        nom[i] = input('Nombre: ')
        pro[i] = float(input('Promedio: '))

def search(nom, x):
    n = len(nom)
    for i in range(n):
        if x == nom[i]:
            return i
    return -1

def test():
    n = int(input('Cantidad de alumnos: '))
    leg = n * [0]
    nom = n * ['']
    pro = n * [0.0]
    read(leg, nom, pro)

    x = input('Ingrese el nombre del estudiante a buscar: ')
    ind = search(nom, x)

    if ind != -1:
        print('El estudiante pedido está registrado en la posición', ind,
              'y sus datos son:')
        print('Legajo:', leg[ind])
        print('Nombre:', nom[ind])
        print('Promedio:', pro[ind])
    else:
        print('No hay un estudiante con ese nombre...')

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

Seleccione una:



a.

Busca en el arreglo de nombres un estudiante con nombre igual a *x*, mediante la función *search()* aplicando búsqueda secuencial. Si tal nombre existe muestra todos sus datos del estudiante, y si no existe se muestra un mensaje avisando de ese hecho.

¡Correcto!



b.

Busca en el arreglo de nombres un estudiante con nombre igual a x , mediante la función `search()` aplicando búsqueda secuencial. Si tal nombre existe muestra el índice de la casilla que lo contenía, y si no existe se muestra un mensaje avisando de ese hecho.



c.

Busca en el arreglo de nombres un estudiante con nombre igual a x , mediante la función `search()` aplicando búsqueda binaria. Si tal nombre existe muestra todos los datos del estudiante, y si no existe se muestra un mensaje avisando de ese hecho.



d.

Busca en el arreglo de nombres un estudiante con nombre igual a x , mediante la función `search()` aplicando búsqueda secuencial. Si tal nombre existe muestra el promedio (y sólo el promedio) de estudiante, y si no existe se muestra el valor `None`.

El programa que sigue crea y carga tres arreglos con los legajos, nombres y promedios de n estudiantes. ¿Qué hace concretamente la función `check()` del programa?

```
__author__ = 'Cátedra de AED'

def read(leg, nom, pro):
    n = len(leg)
    for i in range(n):
        leg[i] = int(input('Legajo: '))
        nom[i] = input('Nombre: ')
        pro[i] = float(input('Promedio: '))

def display_all(leg, nom, pro):
    n = len(leg)
    for i in range(n):
        print('Legajo:', leg[i], end=' ')
        print('Nombre:', nom[i], end=' ')
        print('Promedio:', pro[i])

def check(pro):
    for p in pro:
        if p < 4:
            return True
    return False

def test():
    n = int(input('Cantidad de alumnos: '))
    leg = n * [0]
    nom = n * ['']
    pro = n * [0.0]
    read(leg, nom, pro)

    print()
    display_all(leg, nom, pro)

    if check(pro):
        print('Se ha registrado al menos un estudiante aplazado...')
```

```
    else:  
        print('No se han registrado estudiantes aplazados...')  
  
if name == '__main__':  
    test()
```

Seleccione una:



a.

Retorna *True* si todos los valores del vector *pro* son mayores o iguales a 4, y *False* si al menos un promedio es menor a 4.



b.

Retorna *True* si el arreglo *pro* contiene al menos un valor menor a 4, y retorna *False* si todos los promedios del arreglo son mayores o iguales a 4.

¡Correcto!



c.

Retorna la suma acumulada de todos los promedios del arreglo *pro*.



d.

Retorna *True* si el arreglo *pro* contiene alguna casilla valiendo *None*, y retorna *False* si todos los casilleros son diferentes de *None*.

¿Qué hace la siguiente función en Python?

```
def contar(n):  
    v = n * [0]  
  
    num = int(input('Ingrese un valor entre 0 y' + str(n) + '(con -1  
corta):'))  
    while num != -1:  
        if 0 <= num < n:  
            v[num] += 1  
        else:  
            print('Error. El número debe ser >= 0 y <', n)  
        num = int(input('Ingrese otro valor entre 0 y' + str(n) +  
'(con -1 corta):'))  
  
    return v
```

Seleccione una:



a.

Carga por teclado una secuencia de números, y usa el vector *v* para almacenar esos números.



b.

Carga por teclado una secuencia de números, y usa el vector `v` para acumular esos números.



c.

Carga por teclado una secuencia de números, y usa el vector `v` para contar cuántas veces apareció cada número.

¡Correcto!



d.

Carga por teclado una secuencia de números, y usa el vector `v` para validar que esos números estén dentro del rango [0, n-1].

Suponga que la variable `codigos` tomada como parámetro en la siguiente función se usa para almacenar números entre 0 y 24 que indican en qué idioma están escritos los n libros de un conjunto de libros que se tiene que procesar (por ejemplo, 0: español, 1: inglés, 2: italiano, etc.) Asuma que ese arreglo de códigos fue creado y cargado correctamente antes de ser invocada la función. ¿Qué hace entonces la siguiente función?

```
def procesar(codigos):
    n = len(codigos)
    c = 25 * [0]
    for i in range(n):
        id = codigos[i]
        if 0 <= id <= 24:
            c[id] += 1
```

```
return c
```

Seleccione una:



a.

Usa un vector `c` de números enteros, para contar cuántos libros hay en cada idioma, y luego retorna el valor más alto del vector de conteos.



b.

Usa un vector `c` de números enteros, para contar cuántos libros hay en cada idioma, y luego retorna la conversión a cadena de caracteres del vector de conteos `c`.



c.

Usa un vector `c` de números enteros, para contar cuántos libros hay en cada idioma, y luego retorna el vector original `codigos`.



d.

Usa un vector *c* de números enteros, para contar cuántos libros hay en cada idioma, y luego retorna el vector de conteos *c*.

¡Correcto!

El siguiente programa crea y carga dos arreglos paralelos *destinos* y *montos* con los datos de *n* llamadas telefónicas registradas para un cliente en distintos momentos. En el arreglo *destinos* se almacenan los códigos de los lugares de destino de cada llamada y se asumen que esos destinos se representan con números entre 0 y 24 (por ejemplo: 0: Estados Unidos, 1: Brasil, etc.) El arreglo *montos* almacena el costo de cada llamada. ¿Qué hace exactamente el programa completo al ejecutarse?

```
author__ = 'Cátedra de AED'

def read(destinos, montos):
    n = len(destinos)
    for i in range(n):
        destinos[i] = int(input('Código de destino de la llamada (valor entre
0 y 24 por favor): '))
        montos[i] = float(input('Monto: '))

def process(destinos, montos):
    n = len(destinos)

    s = 25 * [0]
    for i in range(n):
        d = destinos[i]
        if 0 <= d <= 24:
            s[d] += montos[i]

    return s

def display(s):
    print('Listado solicitado de llamadas...')

    m = len(s)
    for i in range(m):
        if s[i] != 0:
            print('Destino:', i, 'Total:', s[i])

def test():
    n = int(input('Cantidad de llamadas: '))
    destinos = n * [0]
    montos = n * [0.0]
    read(destinos, montos)

    s = process(destinos, montos)

    print()
    display(s)

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

Seleccione una:



a.

Muestra una línea única con el monto total acumulado entre todas las llamadas realizadas.



b.

Muestra los montos acumulados en llamadas realizadas para cada uno de los 25 posibles destinos (sólo para los destinos cuyo monto acumulado sea diferente de cero).

¡Correcto!



c.

Muestra la cantidad de llamadas realizadas para cada uno de los 25 posibles destinos (sólo para los destinos cuyo cantidad de llamadas sea diferente de cero).



d.

Muestra los montos acumulados en llamadas realizadas para cada uno de los 25 posibles destinos (incluyendo en el listado los destinos cuyo monto acumulado sea cero).

Fichas Nro. 20 Análisis de Algoritmos

- Cada algoritmo permite resolver un problema particular.
- Existen varias maneras de resolver un problema, algunas alternativas son mejores que otras.
- Algunos ejemplos de problemas estudiados en el cursado:
 - Buscar un valor en el arreglo unidimensional □ Búsqueda secuencial □ Búsqueda binaria
 - Ordenar un arreglo □ Existen muchos algoritmos diferentes
- Entonces si tenemos diversos algoritmos para resolver el problema... ¿cómo podríamos comparar el rendimiento de cada uno para decidir cuál aplicar en una situación en particular?
- Problema: desarrollar un programa que determine la suma de 1 hasta “n” (siendo “n” inicialmente igual a 1000)*.
- Para resolver este problema se pueden plantear dos alternativas de solución.

```
def suma_iterativo(n):
    suma = 0
    for i in range(1, n+1):
        suma+=i
    return suma
```

```
def suma(n):
    return (n/2) * (n + 1)
```

- Algunas resultados de los algoritmos testeados:
 - El tiempo aumenta a medida que aumenta la cantidad de datos.
 - El tiempo de ejecución depende del hardware disponible.

```

Para 100000 datos el Resultado Constante es: 5000050000.0
El tiempo empleado es: 0.0
Para 100000 datos el Resultado Iterativo es: 5000050000
El tiempo empleado es: 0.0260012149810791
-----
Para 1000000 datos el Resultado Constante es: 500000500000.0
El tiempo empleado es: 0.0
Para 1000000 datos el Resultado Iterativo es: 500000500000
El tiempo empleado es: 0.2570149898529053
-----
Para 10000000 datos el Resultado Constante es: 50000005000000.0
El tiempo empleado es: 0.0
Para 10000000 datos el Resultado Iterativo es: 50000005000000
El tiempo empleado es: 2.3101320266723633
-----
```

- Generalmente los parámetros mayormente usados son:
 - El tiempo de ejecución esperado y
 - El espacio de memoria empleado.

La comparación del rendimiento de dichos algoritmos, suele focalizarse en dos situaciones:

- El rendimiento del algoritmo del caso promedio: cuando se presenta la configuración de datos más común, que surge de tomar los datos en un orden estrictamente aleatorio.
- El rendimiento del algoritmo del peor caso: cuando se presenta la configuración de datos mas favorable.

Fundamentos para el Análisis de Algoritmos

- Uno de los fundamentos del análisis de algoritmos es realizar comparaciones de rendimiento entre diversos algoritmos planteados para resolver el mismo problema.
- Y sobre todo disponer de un recurso formal de medición que indique cuan eficiente es un determinado algoritmo respecto de un cierto factor de comparación.
- Los factores de medición posibles son:
 - Tiempo de ejecución esperado
 - Consumo de memoria
 - Complejidad aparente del código fuente: se tiene en cuenta el algoritmo más compacto, claro y sencillo en cuanto al código fuente.
- En el análisis de algoritmos, ya sea se analice el peor caso o el caso promedio, se intenta comparar rendimientos “relativos” entre diferentes algoritmos.
- Y “no” medir en forma numérica y minuciosa el rendimiento de cada uno de ellos.

- Para poder comparar rendimientos de manera “relativa” se busca poder deducir alguna fórmula o expresión aritmética que permita caracterizar el comportamiento de un algoritmo en cuanto algún parámetro de comparación, como el “tiempo de ejecución”.
- Conociendo que fórmula describe mejor a cada algoritmo, se comparan los comportamientos de las funciones representadas por dichas fórmulas.

Notación O

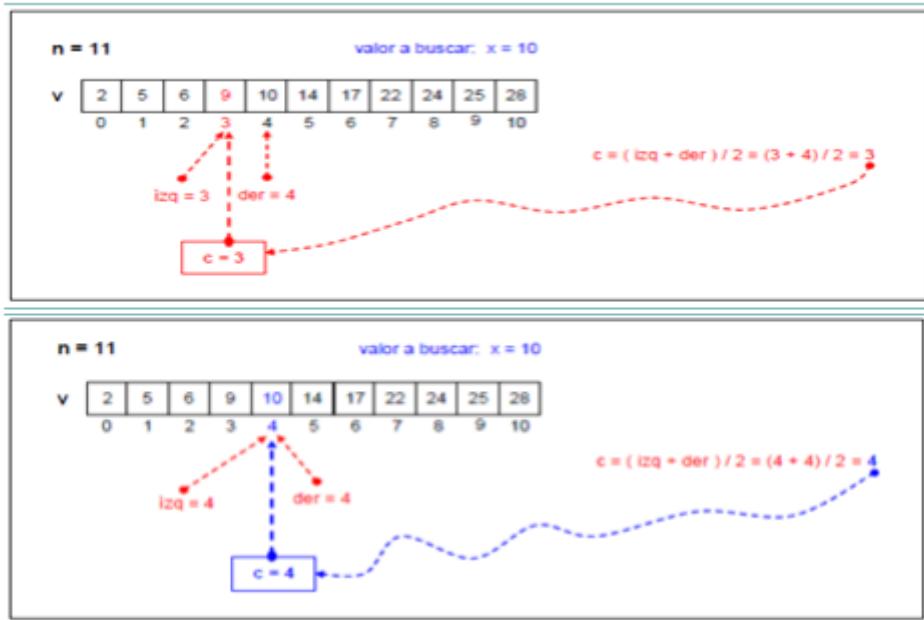
- Es una forma de expresar el rendimiento “relativo” de un algoritmo (ya sea para el tiempo, para el espacio, o para el parámetro que se considere).
- Esta notación se utiliza para indicar el límite superior para el tiempo de ejecución o algún otro parámetro (en realidad la mejor cota superior).
- Se lee como “notación O mayúscula” o como “notación Big O”.
- La notación O describe la complejidad del algoritmo:
 - En términos del tamaño de entrada de datos (n).
 - Independientemente de:
 - Los componentes de hardware.
 - Del lenguaje que se está usando.
 - Y del sistema operativo de base.

Notación O. Búsqueda Secuencial vs Binaria

- Búsqueda Secuencial ◊ Análisis de tiempo de ejecución en peor caso.
- En el peor de los casos se realizan “ n ” comparaciones.
- Se denota simbólicamente: $O(n)$
- En ningún caso demorará más de lo que demore hacer “ n ” comparaciones.

```
def linear_search(v, x):
    # búsqueda secuencial...
    for i in range(len(v)):
        if x == v[i]:
            return i
    return -1
```

- Búsqueda Binaria ◊ Análisis de tiempo de ejecución en peor caso
- Divide el arreglo en 2 hasta que sea necesario.
- Procesa un segmento del arreglo y el otro lo descarta.
- Realiza una comparación en cada partición que se haga del vector.
- La cantidad de veces que podemos dividir por 2 el vector es igual al logaritmo en base 2 de “ n ”.
- Se denota simbólicamente: $O(\log(n))$



Comparando los tiempos de ejecución en el peor de los casos:

Cantidad de datos	Búsqueda Secuencial	Búsqueda Binaria
	$O(n)$	$O(\log(n))$
$n = 1000$	1000 comparaciones	9 a 10 comparaciones
$n = 100.000$	100.000 comparaciones	16 comparaciones

- Selección Directa \diamond Análisis de tiempo de ejecución en peor caso.

- Se realizan $n * n$ comparaciones.
- Se denota simbólicamente: $O(n^2)$
- El tiempo esperado es de orden n^2

```
def selection_sort(v):
    # ordenamiento por selección 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]
```

Funciones de ordenes 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, desechar 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. Recorrido completo de un arreglo.

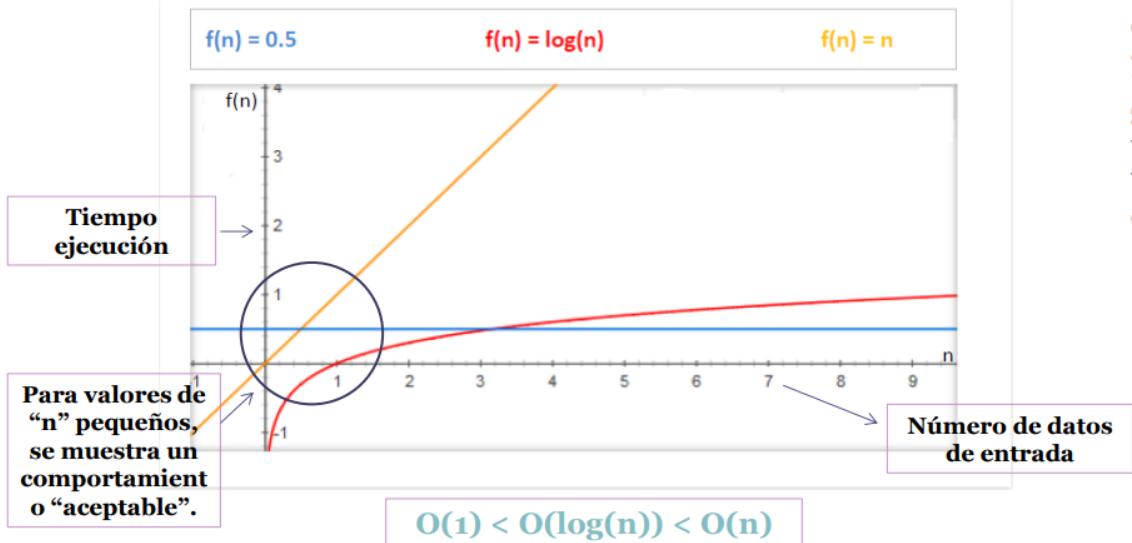
Función	Significado	Casos típicos
$O(n * \log(n))$	Algoritmos que dividen el lote de datos, procesando cada partición sin desechar ninguna, y combinando los resultados al final.	Ordenamiento Rápido (Quick Sort).
$O(n^2)$	Orden cuadrático. Típico de algoritmos que combinan dos ciclos de "n" vueltas cada uno.	Ordenamiento por Selección Directa.
$O(n^3)$	Orden cúbico. Típico de algoritmos que combinan tres ciclos de "n" repeticiones cada uno.	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.

- La forma de crecimiento cuando el "n" es grande o muy grande, se tiene que:

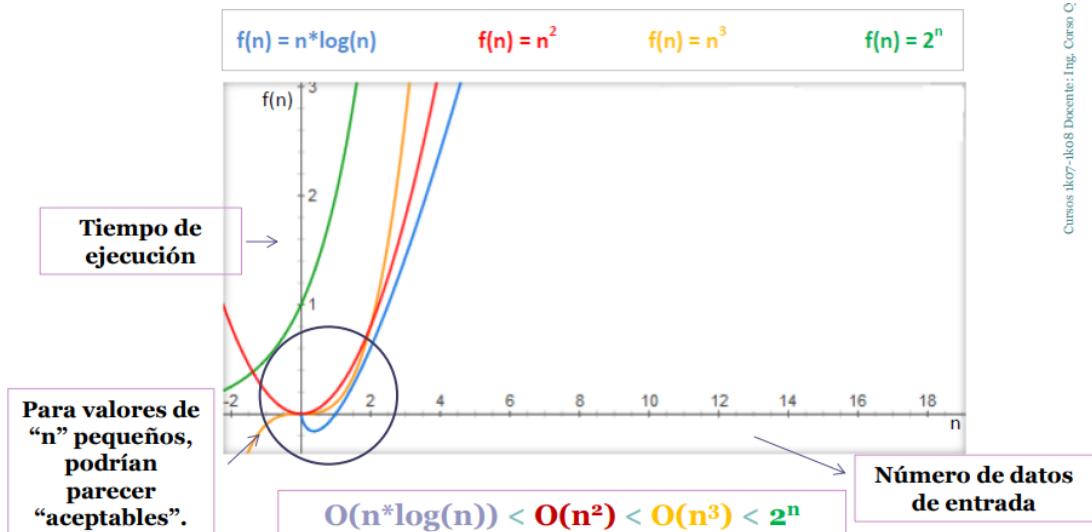
$$O(1) < O(\log(n)) < O(n) < O(n * \log(n)) < O(n^2) < O(n^3) < O(2^n)$$

Formas de crecimiento de las funciones clásicas de complejidad.

- En forma gráfica se puede visualizar el comportamiento de las tres primeras funciones.



- En forma gráfica se puede visualizar el comportamiento de las cuatro últimas funciones.



Análisis de algoritmos. Algunos ejemplos..

- Determinar el orden de complejidad de los algoritmos considerados para resolver el problema inicial: desarrollar un programa que determine la suma de 1 a " n " (siendo n inicialmente igual a 1.000.000)

```
def suma_iterativo(n):
    suma = 0
    for i in range(1, n+1):
        suma+=i
    return suma
```

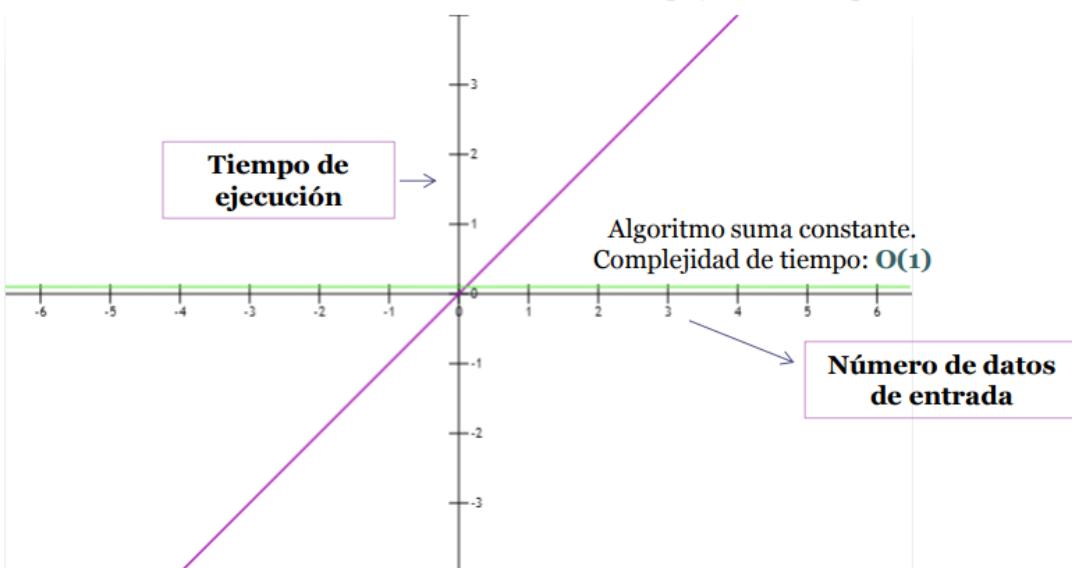
 $\Rightarrow O(n)$

```
def suma(n):
    return (n/2) * (n + 1)
```

 $\Rightarrow O(1)$ 

Cual sería la función de orden de complejidad?

Algoritmo suma lineal.
Complejidad de tiempo: $O(n)$



CUESTIONARIO

Para cada uno de los algoritmos o procesos listados en la columna de la izquierda, seleccione la expresión de orden que mejor describe su tiempo de ejecución en el peor caso.

Dado un arreglo v con n componentes, buscar un valor x aplicando búsqueda secuencial.

Respuesta 1

 $O(n)$

Dado un arreglo v con n componentes, acceder y cambiar el valor del casillero $v[k]$ (con $0 \leq k \leq n-1$).

Respuesta 2

 $O(1)$

Dado un arreglo v con n componentes, ordenarlo de menor a mayor mediante

Respuesta 3

 $O(n^2)$ (léase: "orden n al cuadrado")

el algoritmo de selección directa.

Dadas dos matrices de orden $n \times n$, obtener la matriz producto aplicando el método tradicional.

Respuesta 4

$O(n^3)$ (léase: "orden n al cubo")

Dado un arreglo ya ordenado v con n componentes, buscar un valor x aplicando búsqueda binaria.

Respuesta 5

$O(\log(n))$

¿Cuántas comparaciones en el **peor caso** obliga a hacer una búsqueda secuencial en una *lista ordenada* (o en un *arreglo ordenado*) que contenga n valores?

Seleccione una:

a.

Peor caso: $O(1)$ comparaciones.

b.

Peor caso: $O(\log(n))$ comparaciones.

c.

Peor caso: $O(n^2)$ comparaciones.

d.

Peor caso: $O(n)$ comparaciones.

¡Correcto! Aún estando ordenado el arreglo, en el peor caso una búsqueda secuencial deberá llegar hasta el final si el valor buscado no existe en ese arreglo (o existe y está muy atrás)

¿Cuál es la diferencia entre el **peor caso** y el **caso promedio** en el análisis de algoritmos?

Seleccione una:

a.

El **peor caso** es la configuración de datos de entrada más desfavorable para el algoritmo, mientras que el **caso promedio** describe una configuración de datos pensada para favorecer al algoritmo.

b.

El **peor caso** es la configuración de datos de entrada más favorable para el algoritmo, mientras que el **caso promedio** describe una configuración de datos pensada para desfavorecer al algoritmo.



c.

El **peor caso** es la configuración de datos de entrada más favorable para el algoritmo, mientras que el **caso promedio** describe una configuración aleatoria de datos (no pensada ni para favorecer ni para desfavorecer al algoritmo)



d.

El **peor caso** es la configuración de datos de entrada más desfavorable para el algoritmo, mientras que el **caso promedio** describe una configuración aleatoria de datos (no pensada ni para favorecer ni para desfavorecer al algoritmo).

¡Correcto!

¿Cuáles de los siguientes son **factores de eficiencia comunes** a considerar en el análisis de algoritmos? (Más de una respuesta puede ser válida... marque **todas** las que considere correctas).

Seleccione una o más de una:



a.

La calidad aparente de la interfaz de usuario.



b.

El consumo de memoria.

¡Correcto!



c.

La complejidad aparente del código fuente.

¡Correcto!



d.

El tiempo de ejecución.

¡Correcto!

¿Qué significa decir que un algoritmo dado tiene un tiempo de ejecución **O(1)**?

Seleccione una:



a.

El tiempo de ejecución siempre es de un segundo, sin importar la cantidad de datos.



b.

El tiempo de ejecución es constante, sin importar la cantidad de datos.

¡Correcto!



c.

El tiempo de ejecución es logarítmico: a medida que aumenta el número de datos, aumenta el tiempo pero en forma muy suave.



d.

El tiempo de ejecución es lineal: si aumenta el número de datos, aumenta el tiempo en la misma proporción.

¿Qué significa decir que un algoritmo dado tiene un tiempo de ejecución $O(n^2)$?

Seleccione una:



a.

El tiempo de ejecución es lineal: si aumenta el número de datos, aumenta el tiempo en la misma proporción.



b.

El proceso normalmente consiste en dos ciclos (uno dentro del otro) de aproximadamente n iteraciones cada uno, de forma que las operaciones críticas se aplican un número cuadrático de veces.

¡Correcto!



c.

El tiempo de ejecución es constante, sin importar la cantidad de datos.



d.

A medida que aumenta el número de datos, aumenta el tiempo pero en forma muy suave: el conjunto de datos se divide en dos. se procesa una de las mitades, se desecha la otra y se repite el proceso hasta que no pueda volver a dividirse la mitad que haya quedado.

¿Qué significa decir que un algoritmo dado tiene un tiempo de ejecución $O(n)$?

Seleccione una:



a.

A medida que aumenta el número de datos, aumenta el tiempo pero en forma muy suave: el conjunto de datos se divide en dos. se procesa una de las mitades, se desecha la otra y se repite el proceso hasta que no pueda volver a dividirse la mitad que haya quedado.



b.

El tiempo de ejecución es constante, sin importar la cantidad de datos.



c.

El proceso normalmente consiste en dos ciclos (uno dentro del otro) de aproximadamente n iteraciones cada uno, de forma que las operaciones críticas se aplican un número cuadrático de veces.



d.

El tiempo de ejecución es lineal: si aumenta el número de datos, aumenta el tiempo en la misma proporción.

¡Correcto!

¿Qué significa decir que un algoritmo dado tiene un tiempo de ejecución $O(\log(n))$?

Seleccione una:



a.

El proceso normalmente consiste en dos ciclos (uno dentro del otro) de aproximadamente n iteraciones cada uno, de forma que las operaciones críticas se aplican un número cuadrático de veces.



b.

A medida que aumenta el número de datos, aumenta el tiempo pero en forma muy suave: el conjunto de datos se divide en dos. se procesa una de las mitades, se desecha la otra y se repite el proceso hasta que no pueda volver a dividirse la mitad que haya quedado.

¡Correcto!



c.

El tiempo de ejecución es constante, sin importar la cantidad de datos.



d.

El tiempo de ejecución es lineal: si aumenta el número de datos, aumenta el tiempo en la misma proporción.

Suponga que dispone de cuatro algoritmos diferentes para resolver el mismo problema, y que se sabe que los tiempos de ejecución (en el peor caso) son, respectivamente: $O(n \log n)$, $O(n^2)$, $O(n^3)$ y $O(n)$.

¿Cuál de esos tres algoritmos debería elegir, suponiendo que todos hacen el mismo consumo razonable de memoria?

Seleccione una:



a.

El algoritmo cuyo tiempo de ejecución es $O(n^3)$



b.

El algoritmo cuyo tiempo de ejecución es $O(n \cdot \log(n))$



c.

El algoritmo cuyo tiempo de ejecución es $O(n)$

¡Correcto! Efectivamente, este algoritmo sería el más rápido...



d.

El algoritmo cuyo tiempo de ejecución es $O(n^2)$

¿Con qué nombre general se conoce en la *Teoría de la Complejidad* a un problema para el cual sólo se conocen algoritmos cuyo tiempo de ejecución es exponencial (o sea, problemas para los que todas las soluciones conocidas son algoritmos con tiempo $O(2^n)$)?

Seleccione una:



a.

Problemas Inmanejables



b.

Problemas Intratables

¡Correcto!



c.

Problemas Imperdonables



d.

Problemas Irresolubles

FICHA 21 Formalización de algoritmos

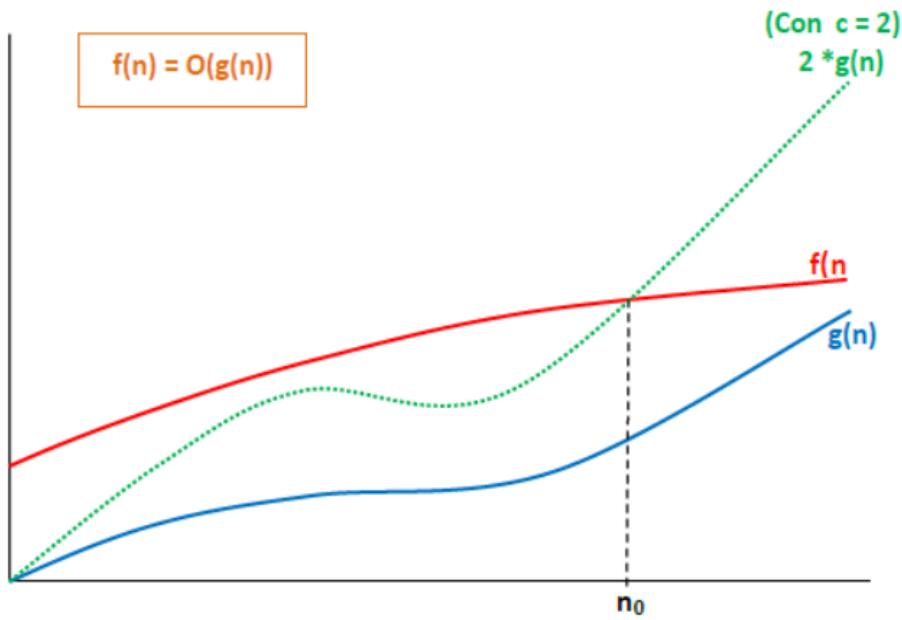
Formalización de la Notación Big O

- La notación Big O se utiliza para indicar un límite o cota superior para el comportamiento de una función.
- Por ejemplo, si un algoritmo de ordenamiento (selección directa) tiene un tiempo de ejecución en el peor caso de $O(n^2)$, de alguna manera se está diciendo que ese algoritmo nunca se comportará peor que n^2 (o alguna función múltiplo de n^2) en cuanto al tiempo de ejecución.
- Se puede decir que una función “f” (que mide el tiempo o espacio de un algoritmo de acuerdo a los valores de “n”) está al 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.

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

Formalización de la notación Big O

Figura 3: Idea general del significado de $f(n) = O(g(n))$.



Observaciones de Notación Big O

- Toda función f polinómica en orden “n” de grado “k”, es de orden n^k . Ejemplos:

-
- Si $f(n) = \frac{1}{2} n^2 + \frac{1}{2} n$ entonces $f(n) = O(n^2)$
 - Si $f(n) = 3n^4 + 2n^2 - 5$ entonces $f(n) = O(n^4)$

No es necesario escribir la función completa en notación Big O: no se debe incluir constantes

- Toda función exponencial en $n + k$, es de orden 2^n . Ejemplo:

$$\text{Si } f(n) = 2^{n+10} \text{ entonces } f(n) = O(2^n)$$

En notación Big O no es necesario incluir las constantes que se sumen a “n” en la función exponencial

- Toda función f logarítmica en n, es de orden $\log_2(n) = \log(n)$. Ejemplo:

$$\text{Si } f(n) = \log_4(n) \text{ entonces } f(n) = O(\log_2(n)) = O(\log(n))$$

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

Consideraciones Prácticas

- Consejos prácticos para intuir una relación de orden (notación Big O) para un algoritmo en el peor caso.
 - a) Identificar cual será el factor de eficiencia para analizar de el algoritmo.
 - b) Determinar con exactitud el tamaño del problema (o volumen de datos a procesar).
 - c) Identificación de la instrucción crítica del algoritmo.
- La instrucción crítica representa la instrucción o bloque de instrucciones que se ejecuta mayor cantidad de veces a lo largo de toda la corrida.
- d) Tener en cuenta!! No es lo mismo realizar:
 - Un conteo exhaustivo y riguroso de operaciones críticas, que hacer y
 - Un análisis asintótico.

Conteo exhaustivo vs Análisis asintótico

- **Conteo exhaustivo y riguroso:** trata de expresar una fórmula con toda rigurosidad y detalle de constantes, términos no dominantes y términos independientes que señalen cuantas operaciones críticas se ejecutan exactamente para un valor “n”. Ejemplo: para el algoritmo de ordenamiento selección directa:

selección 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]
```

Pasada 1:	n-1 comparaciones
Pasada 2:	n-2 comparaciones
Pasada 3:	n-3 comparaciones
...	...
Pasada n-2:	2 comparaciones
Pasada n-1:	1 comparación

Total de comparaciones: $t(n) = (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1.$

Equivale a: $t(n) = ((n-1) * n) / 2$, o bien: $t(n) = 1/2 n^2 - 1/2 n$

- **Análisis asintótico:** se busca identificar la forma general de variación de la función y para ello se aplica una notación conocida como “Big O”. Es decir que se expresa la función pero sin constantes ni términos que no son dominantes cuando “n” tiende a infinito. En el caso del conteo exhaustivo si se lleva a notación Big O para el algoritmo de selección directa: $t(n) = O(n^2)$

Ejemplos de análisis asintótico

- Si la operación crítica (es por ejemplo una comparación) y esta contenida dentro de dos ciclos anidados de n iteraciones cada uno, entonces su algoritmo es de $O(n^2)$.
- Si el algoritmo consta de varios bloques de instrucciones independientes entre sí:

- El algoritmo tendrá un tiempo de ejecución en el orden del que corresponda al bloque con mayor tiempo.

$$t(n) = O(n) + O(n^2) + O(\log(n))$$

Es asintóticamente igual a $O(n^2)$

- Si el algoritmo esta compuesto por un bloque de instrucciones de tiempos constantes, sin ciclos ni procesos dentro de una función, entonces todo el algoritmo se ejecuta en tiempo constante $t(n) = O(1)$.
- Si el algoritmo se basa en tomar un bloque de “n” datos, dividirlo por 2, aplicar una operación de tiempo constante y luego procesar solo una de las mitades de forma de volver a dividirla y continuar así hasta no poder hacer otra división, entonces el algoritmo se ejecuta en tiempo

$t(n) = O(\log_2(n))$ que es lo mismo que

$$t(n) = O(\log(n))$$

(Recordar: no se considera la base del algoritmo)

Para realizar análisis asintótico....

Si consideramos la estructura del siguiente algoritmo..

```
for i in range(n) :  
    for j in range(n) :  
        for k in range(n) :  
            instrucción crítica  
  
for i in range(n) :  
    for j in range(n) :  
        instrucción crítica
```

$$t(n) = O(n^3) + O(n^2)$$



$$t(n) = O(n^3)$$



Cual sería la función de orden de complejidad?

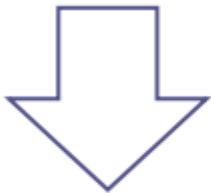
Análisis asintótico en notación Big O en el peor caso

1. Búsqueda secuencial de un valor “x” en un arreglo desordenado de n componentes.



$$t(n) = O(n)$$

2. Búsqueda secuencial de un valor “x” en un arreglo ordenado n componentes.



$$t(n) = O(n)$$

3. Conteo de la cantidad de valores de un arreglo de tamaño n que son mayores al promedio de todos los valores del arreglo.



$$t(n) = O(n) + O(n) = O(2^n)$$



$$t(n) = O(n)$$

4. Buscar un elemento en un vector usando la búsqueda binaria.



$$t(n) = O(\log(n))$$

5. Inserción y eliminación de un elemento en una pila o cola.



$$t(n) = O(1)$$

CUESTIONARIO

Se tiene un algoritmo que realiza cierta cantidad de procesos sobre un conjunto de **n** datos y un minucioso análisis matemático ha determinado que la cantidad de procesos que el algoritmo realiza en el peor caso viene descripto por la función $f(n)$

$= 3n^3 + 5n^2 + 2n^{1.5}$ ¿Cuál de las siguientes expresiones representa mejor el orden del algoritmo para el peor caso?

Seleccione una:



a.

$O(3n^3 + 5n^2 + 2n^{1.5})$



b.

$O(n^{1.5})$



c.

$O(n^3 + n^2)$



d.

$O(n^3)$

¡Correcto!

En la columna de la izquierda se muestra el código fuente en Python de diversos procesos sencillos. Para cada uno de ellos, seleccione de la lista de la derecha la expresión en notación Big O que mejor describa el tiempo de ejecución de cada proceso para el peor caso. (Aclaración: una expresión como n^2 debe entenderse como "n al cuadrado" o n^2).

```
ac = 0
for i in range(10):
    ac += i
print(ac)
~
n = int(input('N: '))
ac = 0
for i in range(n):
    ac += i
print(ac)
~
n = int(input('N: '))
ac = 0
for i in range(n):
    for j in range(i+1, n):
        ac += i*j
for i in range(n):
    for j in range(n):
        for k in range(n):
            ac += (i+j+k)
print(ac)
~
n = int(input('N: '))
ac = 0
```

Respuesta 1

$O(1)$

Respuesta 2

$O(n)$

Respuesta 3

$O(n^3)$

Respuesta 4

$O(n^2)$

```
for i in range(n):
    for j in range(i+1, n):
        ac += i*j
print(ac)
```

Analice el siguiente esquema de una función en Python:

```
def procesar(n, m):
    for i in range(n+1):
        for j in range(m+1):
            # ... acciones sencillas a realizar...
            # ... suponga que no hay otro ciclo
    aquí... # ... y que sólo aparecen operaciones de
            # tiempo constante...
```

¿Cuál de las siguientes expresiones de orden describe mejor el tiempo de ejecución de esta función en el peor caso?

Seleccione una:



- a.
 $O(n^2)$



- b.
 $O(n*m)$

¡Correcto! Efectivamente, no hay problema alguno en que una expresión de orden se base en dos o más variables si el tamaño del problema depende de esas variables.



- c.
 $O(n)$



- d.
 $O(m)$

¿Cuál es la **principal** característica de todos los métodos de ordenamiento conocidos como **métodos simples o directos**?

Seleccione una:



- a.
Son muy fáciles de programar.



- b.
Son muy veloces para cualquier tamaño del arreglo a ordenar.



c.

Tienen un tiempo de ejecución de orden cuadrático.

¡Correcto!

d.

Tienen un tiempo de ejecución de orden lineal.

¿Cuáles de las siguientes son características **correctas** del algoritmo *Shellsort*? (Más de una puede ser cierta... marque TODAS las que considere válidas)

Seleccione una o más de una:

a.

En el caso promedio, el algoritmo *Shellsort* es tan eficiente como el *Heapsort* o el *Quicksort*, con tiempo de ejecución $O(n^*log(n))$.

b.

El algoritmo *Shellsort* es complejo de analizar para determinar su rendimientos en forma matemática. Se sabe que para la serie de incrementos decrecientes usada en la implementación vista en las clases de la asignatura, tiene un tiempo de ejecución para el peor caso de $O(n^{1.5})$.

¡Correcto!

c.

El algoritmo *Shellsort* consiste en una mejora del algoritmo de *Selección Directa*, consistente en buscar iterativamente el menor (o el mayor) entre los elementos que quedan en el vector, para llevarlo a su posición correcta, pero de forma que la búsqueda del menor en cada vuelta se haga en tiempo logarítmico.

d.

El algoritmo *Shellsort* consiste en una mejora del algoritmo de *Inserción Directa* (o *Inserción Simple*), consistente en armar suconjuntos ordenados con elementos a distancia $h > 1$ en las primeras fases, y terminar con $h = 1$ en la última.

¡Correcto!

¿Qué diferencia existe entre el **conteo exhaustivo** de operaciones críticas y el **análisis asintótico** del comportamiento de una función en el análisis de algoritmos?

Seleccione una:

a.

Ninguna. Ambas se refieren a la misma técnica básica del análisis de algoritmos



b.

El **conteo exhaustivo** busca determinar una expresión o fórmula que exprese de manera rigurosa la cantidad de operaciones críticas que lleva a cabo un algoritmo, mientras que el **análisis asintótico** busca determinar el comportamiento general de una función para valores muy grandes del tamaño del problema.

¡Correcto!



c.

El **análisis asintótico** busca determinar una expresión o fórmula que exprese de manera rigurosa la cantidad de operaciones críticas que lleva a cabo un algoritmo, mientras que el **conteo exhaustivo** busca determinar el comportamiento general de una función para valores muy grandes del tamaño del problema.



d.

El **conteo exhaustivo** busca determinar una expresión o fórmula que exprese de manera rigurosa la cantidad de operaciones **críticas** que lleva a cabo un algoritmo, mientras que el **análisis asintótico** busca determinar una expresión o fórmula que exprese de manera rigurosa la cantidad de operaciones **no críticas** que lleva a cabo un algoritmo.

¿Qué se entiende, en el contexto del Análisis de Algoritmos, por un *Orden de Complejidad*?

Seleccione una:



a.

Un conjunto o familia de funciones matemáticas que se comportan asintóticamente de la misma forma.

¡Correcto!



b.

Un conjunto o familia de algoritmos que resuelven el mismo problema.



c.

Un conjunto o familia de subrutinas con similares objetivos (equivalente al concepto de *módulo*).



d.

Un conjunto de datos ordenados.

Si los algoritmos de *ordenamiento simples* tienen todos un tiempo de ejecución $O(n^2)$ en el peor caso, entonces: ¿cómo explica que las mediciones efectivas de los tiempos de ejecución de cada uno sean diferentes frente al mismo arreglo?

Seleccione una:



a.

La notación *Big O* rescata el término más significativo en la expresión que calcula el rendimiento, descartando constantes y otros términos que podrían no coincidir en los tres algoritmos.

¡Correcto!



b.

Los tiempos deben coincidir. Si hay diferencias, se debe a errores en los instrumentos de medición o a un planteo incorrecto del proceso de medición.



c.

La notación *Big O* no se debe usar para estimar el comportamiento en el peor caso, sino sólo para el caso medio.



d.

La notación *Big O* no se usa para medir tiempos sino para contar comparaciones u otro elemento de interés. Es un error, entonces, decir que los tiempos tienen "*orden n cuadrado*".

¿Cuáles de las siguientes *son correctas* en cuanto a los tiempos de ejecución de los algoritmos de ordenamiento clásicos? (Más de una puede ser cierta... marque TODAS las que considere válidas)

Seleccione una o más de una:



a.

Algoritmos directos o simples: $O(n^2)$ en el peor caso para todos ellos.

¡Correcto!



b.

Algoritmo Shell Sort: $O(n^2)$ en el peor caso para la serie de incrementos decrecientes vista en clase.



c.

Algoritmo Heap Sort: $O(n \log n)$ tanto para el caso promedio como para el peor caso.

¡Correcto!



d.

Algoritmo Quick Sort: $O(n \log n)$ en el caso promedio, pero $O(n^2)$ en el peor caso.

¡Correcto!

Si se realiza un análisis preciso del ordenamiento por *Selección Directa* para un arreglo de n componentes, se llega a la conclusión que ese algoritmo hará $n-1$ pasadas, con $n-1$ comparaciones en la primera, $n-2$ en la segunda, y así sucesivamente reduciendo de a 1 la cantidad de comparaciones hasta hacer sólo una comparación en la última pasada. Por lo tanto, el algoritmo hará *invariablemente* una cantidad total de $\frac{1}{2}(n^2 - n)$ comparaciones. Sabiendo esto, ¿cuáles de las siguientes expresiones son correctas para describir la cantidad de comparaciones que hará el algoritmo, usando distintos tipos de notaciones? (Más de una respuesta puede ser correcta. Marque TODAS las que considere correctas)

Seleccione una o más de una:



a.

Cantidad de comparaciones: $\Omega(n^2)$

¡Correcto!



b.

Cantidad de comparaciones: $O(n^2)$

¡Correcto!



c.

Cantidad de comparaciones: $o(n^2)$



d.

Cantidad de comparaciones: $\Theta(n^2)$

¡Correcto!

FICHA 22 Archivos

Archivos: Introducción

- Es una estructura de datos lineal almacenada en dispositivos de almacenamiento externos (como discos, pendrive, DVD etc.).
- El almacenamiento de los datos es permanente (no volátil) sin perderlos cada vez que el programa finaliza.
- Permite la gestión de grandes volúmenes de datos.
- Pueden ser utilizados por cualquier otro programa.
- Desde cualquier programa es posible agregar, eliminar y modificar datos.
- Los archivos almacenan sus datos en sistema binario.
- Cada dato almacenado utiliza tantos bytes como sea necesario para representar ese dato. Por lo tanto el tamaño de un archivo está determinado por la “cantidad de bytes” que contiene.
- En la práctica, cabe hacer una diferenciación entre:

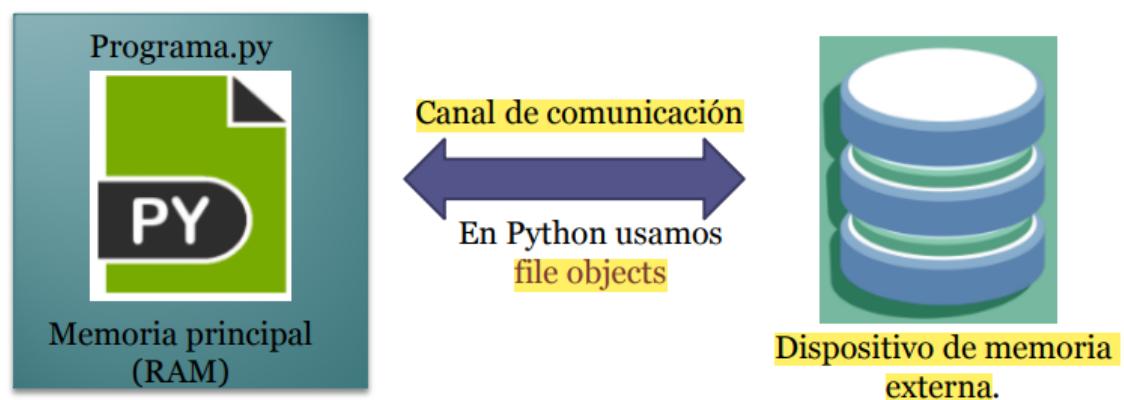
• Archivos de **texto**: todos los bytes del archivo son interpretados y visualizados como caracteres (por ejemplo de acuerdo a la tabla de código ASCII).

• Archivos **binarios**: Las secuencias o bloques de bytes que el archivo contiene representan información de cualquier tipo (caracteres, valores booleanos, etc.) y no asume que cada byte representa un carácter.

Operaciones básicas: Apertura de archivo

- La función `open()` es la encargada de abrir el canal de comunicación entre el dispositivo que contiene al archivo y la memoria principal.
- El objeto creado y retornado por la función (`file object`) se aloja en memoria y contiene diversas funciones que permiten manejar el archivo (manejador del archivo).

La sintaxis es: `open(nombre físico, modo de apertura)`



Ejemplo de modo de apertura de un archivo:

```
m = open("datos.dat", "wb")
```

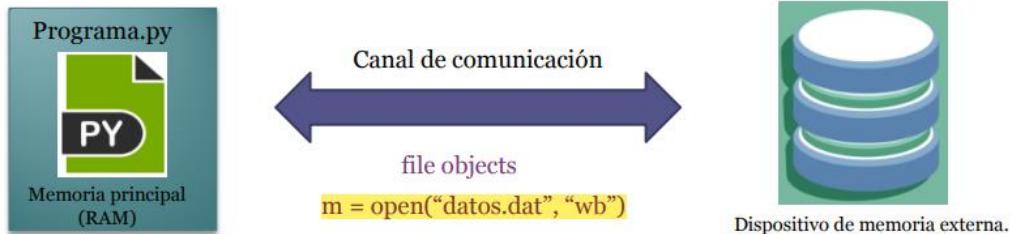
El archivo se busca en la carpeta actual del proyecto o programa que se este ejecutando.

```
# crea una variable m asociada a un file object y  
# deja abierto el archivo datos.dat en modo de  
# grabación.
```

Otra variante de apertura de archivo válida:

```
m = open("c:\\files\\datos.dat", "wb")
```

Se especifica el directorio de acceso del archivo.



Modos Apertura de archivos binarios

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.

Operaciones básicas: Cierre del archivo

- La forma de cierre de los archivos se puede realizar de dos formas:

• **Automático:** se presenta cuando la variable (file object) para acceder al archivo sale del ámbito donde esta definida.

• Manual: invocando la función close(). Ejemplo:

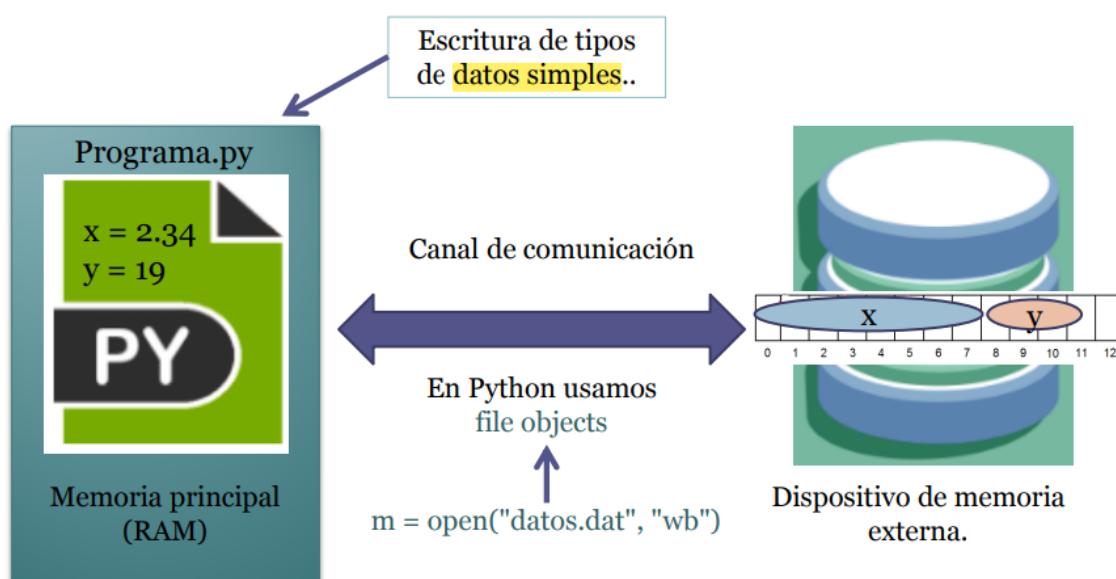
```
m = open('datos.dat', 'wb')
m.close()
```

- El programador debe estar seguro que el archivo que estuvo manejando se cierre oportunamente, ya sea en forma manual o en forma automática.



Operaciones básicas: Lectura y escritura

- Los objetos de tipo file (file object) que se crean con open() contienen numerosos funciones adicionales para el manejo de archivos.
- En el caso de los archivos de textos se dispone:
 - ♣ Para la lectura de datos: read()
 - ♣ Para la escritura de datos: write()
- Ambas son directas y simples de usar cuando se trata de archivos de textos, pero no son tan directas cuando se trata de un archivo de tipo binario.
- En el caso de los archivos binarios se puede hacer:
 - ♣ Lectura y grabación de datos simples: entero, float entre otros.
 - ♣ Lectura y grabación de datos compuestos: registros, listas etc.



- El proceso de lectura y escritura se realiza mediante un proceso denominado serialización.
- La “serialización” es un proceso por el cual el contenido de una variable normalmente de estructura compleja 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.
- Para el mecanismo de serialización en Python se usará el módulo pickle. Este módulo provee ciertas funciones:
 - ♣ Para la escritura: dump()
 - ♣ Para la lectura: load()

Operaciones básicas: **escritura (tipo simple)**

- Para grabar los datos en un archivo se hace mediante la función dump().
- Ejemplo como grabar datos simples en un archivo binario.

```
import pickle

print('Procediendo a grabar números en el archivo')

m = open('prueba.dat', 'wb')
x, y = 2.345, 19

pickle.dump(x, m)
pickle.dump(y, m) ←
m.close()

dump(arg1,arg2):
  • 1 arg = el valor a grabar.
  • 2 arg = manejador al archivo (file object)
```

Operaciones básicas: **lectura (tipo simple)**

- Para leer los datos en un archivo se hace mediante la función load().
- Ejemplo como leer datos simples en un archivo binario

```
import pickle

print('Procediendo a leer números en el archivo')

m = open('prueba.dat', 'rb')

a = pickle.load(m)
b = pickle.load(m) ←
m.close()

print('Datos recuperados:', a, '-', b)

load (arg1)
  • 1 arg= manejador al archivo (file object)
```

Operaciones básicas: Lectura y escritura (tipo compuesto)

- Lectura y grabación de una variable registro en un archivo binario. Ejemplo: Guardar en el archivo tres variables de tipo registro Libro.

```
import pickle

class Libro:
    def __init__(self, cod, tit, aut):
        self.isbn = cod
        self.titulo = tit
        self.autor = aut

def display(libro):
    print('ISBN:', libro.isbn, end ='')
    print('Titulo:', libro.titulo, end ='')
    print('Autor:', libro.autor)
```

- Lectura y grabación de una variable registro en un archivo binario. Ejemplo: Guardar en el archivo tres variables de tipo registro Libro.

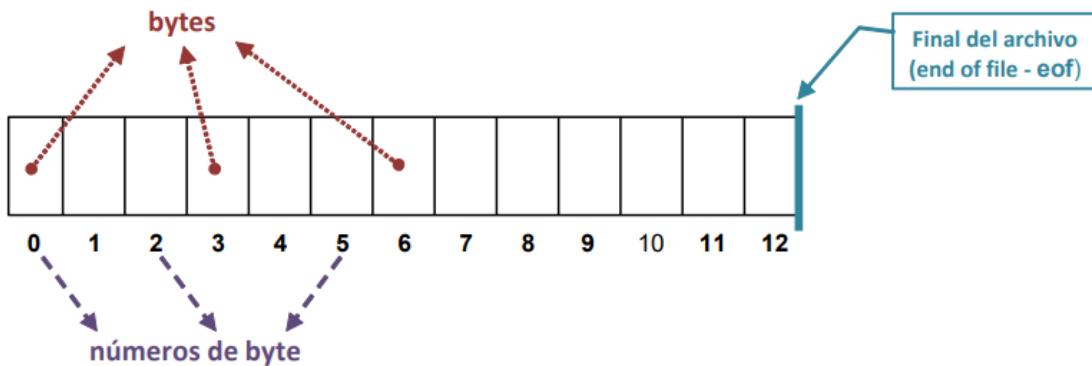
```
def test:
    print('Prueba de grabación de registro')
    lib1 = Libro(123, 'Libro1', 'Autor1')
    lib2 = Libro(234, 'Libro2', 'Autor2')
    fd = 'libros.dat'
    m = open(fd, 'wb')
    pickle.dump(lib1,m)
    pickle.dump(lib2,m)

    lib1 = pickle.load(m)
    lib2 = pickle.load(m)
    m.close()
    display(lib1)
    display(lib2)

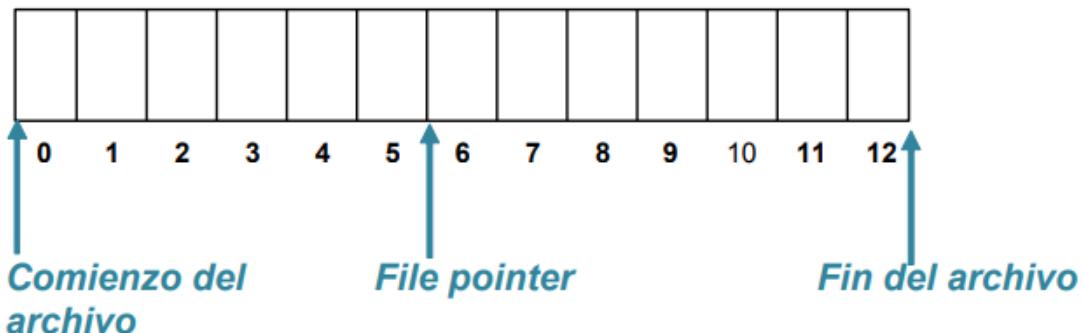
if __name__ == '__main__':
    test()
```

El objeto File Pointer (i)

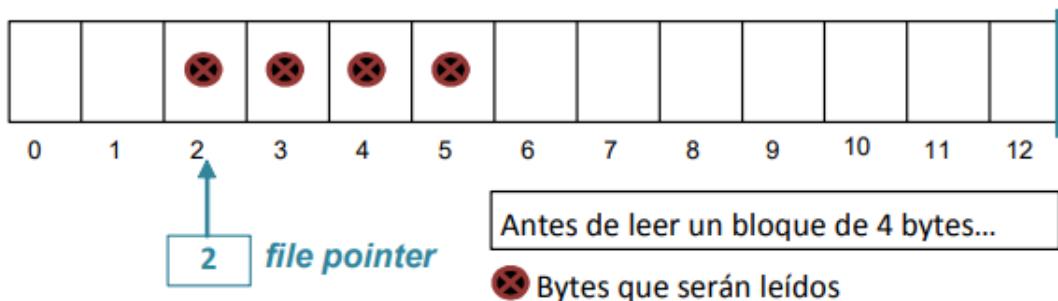
- La estructura de un archivo puede entenderse con un arreglo de bytes en memoria externa. Cada componente de este arreglo es uno de los bytes grabados en el archivo, y cada byte tiene a modo de índice un número que lo identifica, correlativo de cero en adelante.



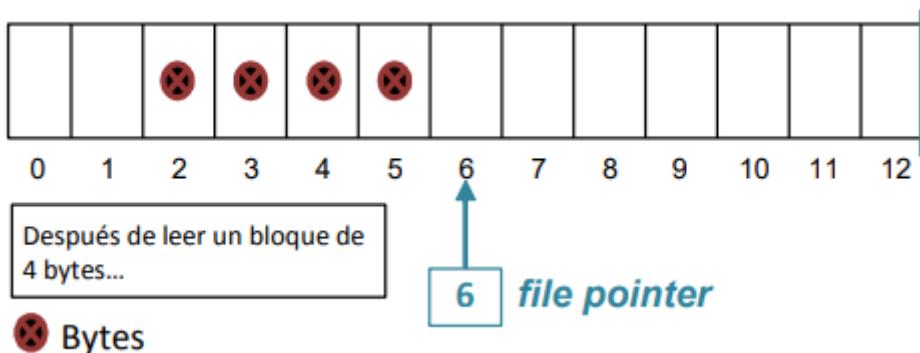
- File Pointer: cursor o indicador o marcador interno, es una de tipo entero que contiene el número del byte sobre el cual se realizará la próxima operación de lectura o de grabación.
- Comienzo del archivo: Es el byte cero, el primer byte del archivo.
- Fin del archivo: marca de final del archivo o end of file (eof).



- Esquema del estado del File Pointer en un archivo antes de proceder la lectura.



- Esquema del estado del File Pointer en un archivo después de proceder la lectura.



- Es gestionado automáticamente por las siguientes funciones:
 - open() ↓ file pointer queda apuntado al inicio del archivo. Es decir en el byte cero.
 - En un proceso de serialización (lectura-grabación):
 - ♣ dump() ↓ file pointer queda apuntando en el byte que indica final del registro grabado.
 - ♣ load() ↓ file pointer queda apuntando en el byte que indica el final del registro leído.

Funciones útiles para recorrer un archivo de manera secuencial

- Para obtener el el tamaño en bytes de un archivo, se utiliza la función getsize() incluida en el módulo os.path:

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

- La función os.path.getsize() toma como parámetro el nombre físico del archivo y retorna su tamaño en bytes.
- No es necesario que el archivo este abierto con la función open() para usar dicha función
- La función tell() retorna la ubicación actual del “file pointer” en forma de un número entero (número de byte).

- **Ejemplo:**

```
m = open('prueba.dat', 'wb')
pos = m.tell()
print('El valor del file pointer: ', pos)
```

↑
El valor del file pointer es 0
(cero)

- Es muy útil cuando por algún motivo se quiere conocer en qué byte está posicionado un archivo en un momento dado.

Archivos binarios: recorrido secuencial

- Para grabar un registro en un archivo usamos dump()

```
fd = 'libros.dat'
m = open(fd, 'ab')
pickle.dump(lib1, m)
pickle.dump(lib2, m)
pickle.dump(lib3, m)
pickle.dump(lib4, m)
pickle.dump(lib5, m)
m.close()
print('Se grabaron varios registros en el archivo', fd)
```

```
m = open(fd, 'rb')
t = os.path.getsize(fd)

print('Se recuperaron estos registros desde el archivo', fd, ':')
while m.tell() < t:
    lib = pickle.load(m)
    display(lib)
m.close()
```

Chequea que el valor actual del puntero sea menor que el tamaño total.

Archivos binarios: acceso directo

- La función seek() permite cambiar el valor del file pointer, y elegir cuál será el próximo byte que será leído o grabado.
- Para poder ser usado debe incluirse en el programa: import io.
- Y se supone que el archivo debe estar previamente abierto.

Forma general: seek(offset, from_what)

El primer parámetro indica cuántos bytes debe moverse el file pointer.

- El segundo parámetro indica desde qué lugar (número de bytes) se hace ese salto
- 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.

- Permiten mover el file pointer para leer o modificar bytes en una posición definida.
- Ejemplos para cambiar el valor de file pointer (puntero). En este caso consideramos que la variable manejadora del archivo se llama "m"

■ Parados al comienzo del archivo (byte 0) y saltar al valor 10:

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

■ Si el valor del file pointer es 7 y se quiere saltar al byte 4:

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

CUESTIONARIO

¿Cuál de las siguientes afirmaciones es **FALSA** respecto de la serialización en Python?

Seleccione una:

a.

El módulo *pickle* es uno de los que brindan funciones para serializar en Python, pero no es el único (existen otros, tales como *json*).

b.

El método del módulo *pickle* que permite grabar una variable directamente es *pickle.dump()* y el que permite recuperar una variable serializada es *pickle.load()*.

c.

La *serialización* es un proceso por el cual el contenido de una variable se convierte automáticamente en una secuencia de bytes listos para ser almacenados en un archivo, y luego recuperarse desde el archivo y volver a crear la variable original.



d.

En Python no se pueden serializar variables de tipo simple (variables de tipo int, float, bool, etc.)

¡Correcto! efectivamente, esta es la afirmación falsa: en Python es perfectamente posible serializar valores de tipo simple y hemos mostrado ejemplos en el proyecto F[21] Archivos.

¿Cuál de las siguientes afirmaciones es **CIERTA** respecto de la operación para obtener el tamaño en bytes de un archivo en Python?

Seleccione una:



a.

La función `os.path.getsize()` toma como parámetro el nombre físico de un archivo, y retorna la longitud en bytes de ese archivo.

¡Correcto!



b.

La función `open()` toma como parámetro el nombre físico de un archivo y el modo de apertura deseado, abre el archivo y retorna el tamaño en bytes del mismo.



c.

En Python no hay forma de obtener el tamaño en bytes de un archivo.



d.

El método `tell()` de los objetos manejadores de archivos, no toma parámetro alguno y retorna siempre el tamaño en bytes del archivo para el cual fue invocado.

¿Cuál de las siguientes es claramente falsa respecto de las características y propiedades de un objeto para manejar archivos en Python (un *file object*, tal como lo crea y lo retorna la función `open()`)?

Seleccione una:



a.

Un *file object* (o un *file-like object*) representa archivos en los que es posible acceder en forma directa a cualquiera de sus bytes mediante el método *seek()*.



b.

Un *file object* (o un *file-like object*) en última instancia permite interpretar el contenido de un archivo como si se tratase de un arreglo de bytes en memoria externa.



c.

Un *file object* (o un *file-like object*) contiene métodos que permiten tanto grabar como leer datos del archivo representado, independientemente de que eso también puede hacerse con funciones de serialización incluidas en módulos separados (como *pickle* o *json*).



d.

Un *file object* (o un *file-like object*) representa sólo archivos de texto (y nunca archivos binarios).

¡Correcto! efectivamente, esta afirmación es falsa: un file object representa un archivo. Da igual si luego se usa como archivo de texto o binario.

¿Cuál de las siguientes es **FALSE** respecto del *file pointer* en un *file object*?

Seleccione una:



a.

La **única forma** en que puede cambiar el valor del *file pointer* de un archivo es invocando al método *seek()* del *file object* que lo maneja.

¡Correcto! Efectivamente, esta afirmación es falsa. El valor del file pointer puede cambiar usando *seek()*, pero esa no es la única forma: su valor inicial es establecido por *open()* y luego también cambia de valor cada vez que realiza una lectura o una grabación.



b.

Cada vez que termina una operación de lectura o grabación en el archivo, el *file pointer* queda apuntando al byte siguiente al último que se leyó o grabó.



c.

El valor inicial del *file pointer* es asignado por la función *open()* al abrir el archivo.



d.

El *file pointer* de un archivo puede apuntar a algún byte que esté fuera del archivo, *a la derecha del final del archivo*, pero no puede contener un valor negativo.

Si bien sabemos que todo archivo contiene datos representados en binario (y en ese sentido, todo archivo es un archivo binario), el hecho es que los archivos en general se clasifican en *archivos binarios* y *archivos de texto*. ¿Cuál es la diferencia entre ambos?

Seleccione una:



a.

Todos los archivos son binarios en cuanto a su contenido. La distinción se hace en cuanto a cómo se **interpreta** ese contenido. En los archivos de texto, todos los bytes se interpretan como caracteres visualizables, salvo los saltos de línea y/o retornos de carro. Pero en los archivos binarios, cada byte puede representar cualquier tipo de dato. La interpretación depende del programa que procese a ese archivo.

¡Correcto!



b.

No hay ninguna diferencia. Todos los archivos son binarios, y esa distinción es incorrecta.



c.

No es cierto que todos los archivos son binarios. Algunos contienen efectivamente texto, otros contienen imágenes, otros contienen números, y en definitiva pueden contener datos del tipo que sea.



d.

Un archivo de texto sólo (en la plataforma *Windows*) debería contener bytes cuyos valores sean mayores a 31 (caracteres visualizables), y **sólo bytes mayores a 31**. La presencia de cualquier byte con valor menor a 32, lleva a que ese archivo sea considerado binario (por la presencia de bytes que no representan caracteres visualizables).

Analice el siguiente script Python, en el cual se están grabando registros de tipo *Libro* en un archivo por serialización:

```
__author__ = 'Catedra de AED'

import pickle

class Libro:
    def __init__(self, cod, tit, aut):
        self.isbn = cod
        self.titulo = tit
        self.autor = aut

def display(libro):
    print('ISBN:', libro.isbn, end='')
```

```

print(' - Título:', libro.titulo, end=' ')
print(' - Autor:', libro.autor)

def test():
    print('Prueba de grabación de varios registros...')
    lib1 = Libro(2134, 'Fundación', 'Isaac Asimov')
    lib2 = Libro(5587, 'Fundación e Imperio', 'Isaac Asimov')
    lib3 = Libro(3471, 'Segunda Fundación', 'Isaac Asimov')

    fd = 'libros.dat'
    m = open(fd, 'wb')
    pickle.dump(lib1, m)
    pickle.dump(lib2, m)
    m.close()
    print('Se grabaron varios registros en el archivo', fd)

    m = open(fd, 'rb')
    lib1 = pickle.load(m)
    lib2 = pickle.load(m)
    lib3 = pickle.load(m)
    m.close()

    print('Se recuperaron estos registros desde el archivo', fd, ':')
    display(lib1)
    display(lib2)
    display(lib3)

if __name__ == '__main__':
    test()

```

¿Hay algún problema con este script?

Seleccione una:



a.

Sí. El problema es que no se pueden grabar registros en un archivo si el mismo fue abierto con *open()*. Debió usar *open_register_file()*.



b.

Sí. El problema es que el archivo fue abierto en modo *wb* cuando se pretendió grabar, pero en ese modo no se puede grabar.



c.

No hay nada de malo en este segmento.



d.

Sí. Se grabaron sólo dos registros en el archivo, pero luego se intentaron tres lecturas. La tercera fallará por encontrar el final del archivo en forma inesperada.

¡Correcto!

¿Qué ocurre si se intenta leer en un archivo y el *file pointer* del mismo está apuntando en ese momento al final del archivo (o sea, al primer byte ubicado fuera del archivo)?

Seleccione una:



a.

No ocurre nada. La lectura no se realiza, pero tampoco se lanza un error. El programa simplemente ignora la orden y continúa.



b.

Se leen los bytes que siguen al final del archivo, bajo responsabilidad del programador, y el programa continúa normalmente.



c.

Se lanza un error de intérprete de la forma *EOFError*, y el programa se interrumpe.

¡Correcto!



d.

La pregunta no tiene sentido: el *file pointer* **no puede** estar apuntando al final del archivo.

¿Qué ocurre si en un programa Python se usa el método *seek()* de un *file object* y se salta con ese método a un byte que está más allá del final del archivo?

Seleccione una:



a.

Se interrumpe el programa lanzando un error de la forma *EOFError*.



b.

El método *seek()* no puede saltar a un byte que esté fuera del archivo, por lo tanto, no hará nada.



c.

El *file pointer* del archivo quedará posicionado en ese byte, aunque esté fuera del archivo.

¡Correcto!



d.

El *file pointer* quedará posicionado en ese byte, y el tamaño del archivo se ajustará a ese byte.

Suponga que en un programa en Python necesita abrir un archivo de tal forma que:

- El archivo se maneje como archivo binario (y no como archivo de texto).
- Si el archivo no existía NO sea creado.
- Se permita tanto leer como grabar su contenido.
- No se destruya el contenido si el archivo ya existía.
- Se pueda leer en cualquier posición del archivo y también grabar en cualquier posición del mismo.

¿Cuál de los posibles modos de apertura disponibles para `open()` tendría que emplear al abrir ese archivo?

Seleccione una:



a.

El modo rb.



b.

El modo a+b.



c.

El modo r+t.



d.

El modo r+b.

¡Correcto!

¿Qué valor tendrá el *file pointer* del archivo representado por `m` luego de terminar de realizar las operaciones que se indican? (ignore cualquier situación de error que podría producirse al abrir el archivo o al hacer las grabaciones: sólo analice las líneas dadas suponiendo un contexto correcto):

```
import pickle
import os.path

a = 25
b = 2.876
c = 'Hola mundo'

m = open('prueba.dat', 'wb')
pickle.dump(a, m) # suponer que aquí se grabaron 5 bytes...
pickle.dump(b, m) # suponer que aquí se grabaron 12 bytes...
pickle.dump(c, m) # suponer que aquí se grabaron 20 bytes..
m.close()
```

Seleccione una:



a.

20



b.

37

¡Correcto!



c.

32



d.

5

FICHA 23 Archivos: Gestión ABM

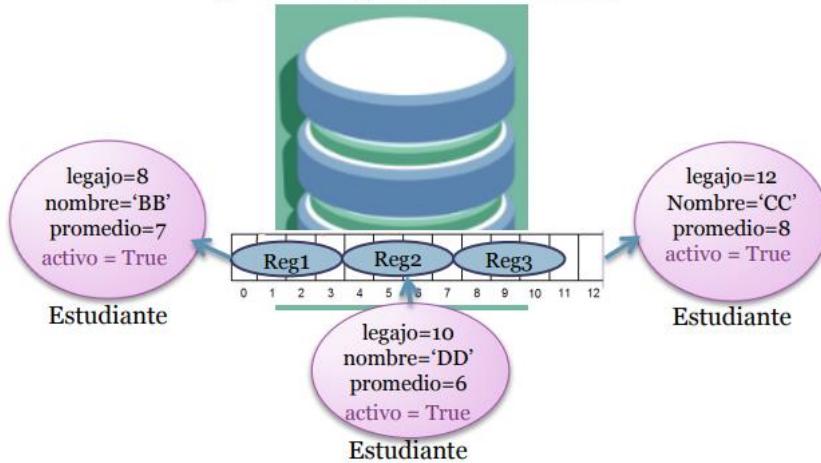
- Gestión de ABM: representan las operaciones que se realizan con los registros de un archivo
 - Incorporación de un registro: Alta
 - Eliminación de un registro: Baja
 - Modificación de un registro: Modificación

Gestión de ABM: Alta de Registros

- Mecánica para el Alta de un registro en el archivo (claves no repetidas)
 1. Con un ciclo se recorre el archivo y en cada iteración se consulta su clave con la que ingresó el usuario para dar de alta.
 2. Si hay coincidencia de clave y no se encuentra marcado como eliminado, entonces el “alta se rechaza” \diamond significa que ya existe.
 3. Si se llega al final del archivo y no se encontró repetida la clave, se “acepta el alta” (asignando a True su marcador lógico).
 4. Se graba el registro al final del archivo.

Gestión de ABM: Alta sin “claves duplicadas”

1) Archivo Original (**estudiantes.dat**)



2) Alta de un estudiante ◊ Se desea agregar un estudiante con un legajo repetido◊ legajo=10

Paso a: Se recorre el archivo (estudiantes.dat) y en cada vuelta se consulta por el legajo ingresado.

Paso b: En el caso de encontrar el legajo y no se encuentra marcado como borrado (activo= True), el “alta se rechaza”.

Paso c: Si se llega al final del archivo y el legajo no se encontró, se procede al “alta” y se coloca el campo activo en True.

Búsqueda de un registro en el archivo

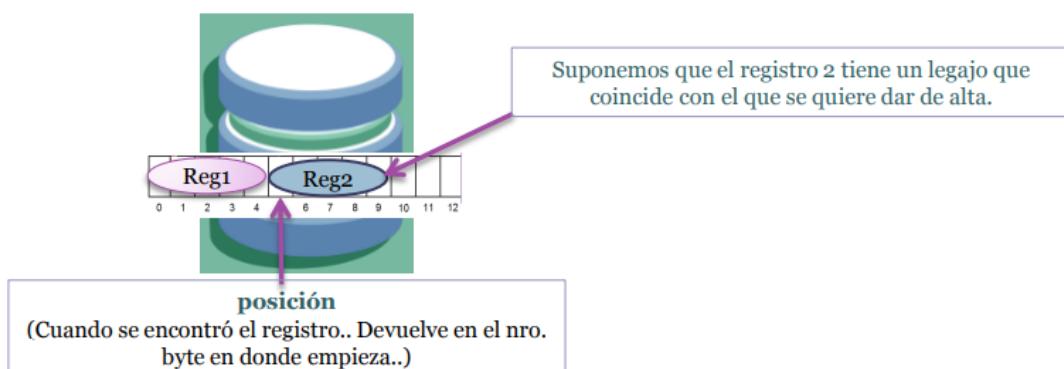
Mecánica para el Búsqueda de un determinado registro.

1. Ingresar la clave de identificación del registro que se quiere buscar. Ejemplo: legajo del estudiante.

2. Búsqueda del registro con esa clave en el archivo.

3. Si no se encuentra informar al usuario, por ejemplo, con el retorno de -1.

4. En el caso de encontrarse, detener la búsqueda y retornar el número de byte interno del archivo (File Pointer) en el cual comienza el registro encontrado.

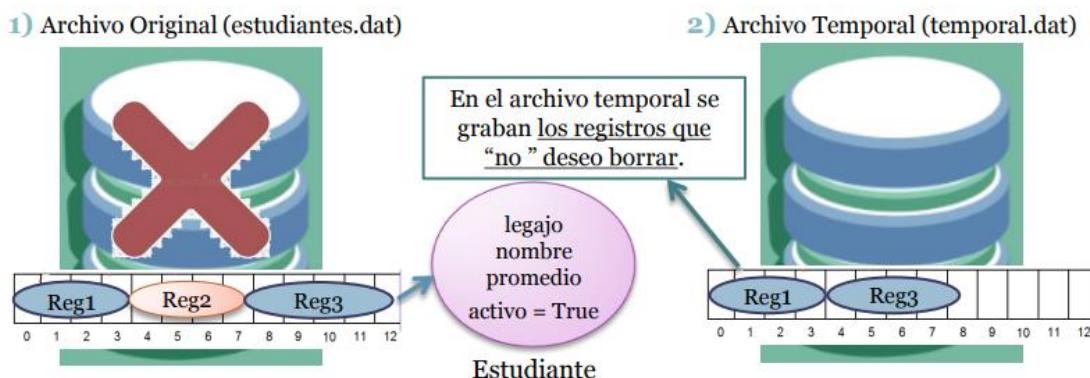


Gestión de ABM: Baja de registros

Mecánica para la “baja” de registro

- El usuario ingresa la clave de identificación del registro a borrar (en el caso del ejemplo: legajo del estudiante). Se procede a la búsqueda si se encuentra el registro se procede a la baja.
- No existe ninguna operación específica para la eliminación de un determinado registro en Python.
- Hay dos vías de solución clásica para tal fin:
 - Baja Física
 - Baja Lógica
- Mecánica para la “baja” física en un archivo
 1. Usar un segundo archivo (temporal).
 2. El archivo original se abre en modo lectura ('rb').
 3. El archivo temporal en modo grabación ('wb').
 4. Con un ciclo se recorre el archivo original hasta el final, verificando si el campo clave (legajo del estudiante) coincide con el valor buscado.
 5. Si el valor no coincide se graba el registro en el archivo temporal. (El único que no se copia es el que se quiere eliminar)
 6. Eliminación del archivo original
 7. Renombra el archivo temporal con el nombre que tenía el archivo original.

Gestión de ABM: Baja Física



3) Eliminar el estudiante correspondiente al =>“registro 2.”

- Paso a: Abrir archivo original modo lectura ('rb').
Paso b: Ingresar por teclado el campo clave (legajo a buscar)
Paso c: Recorrer el archivo original hasta el final, buscando si el campo clave (legajo estudiante) coincide con algún registro y el campo activo = True.
Paso d: En el archivo temporal ('wb') se graban todos los registros que no coincidan con el legajo ingresado por teclado (todos aquellos que no quiero borrar).
Paso e: Eliminar el archivo original.
Paso f: Renombrar el archivo temporal.

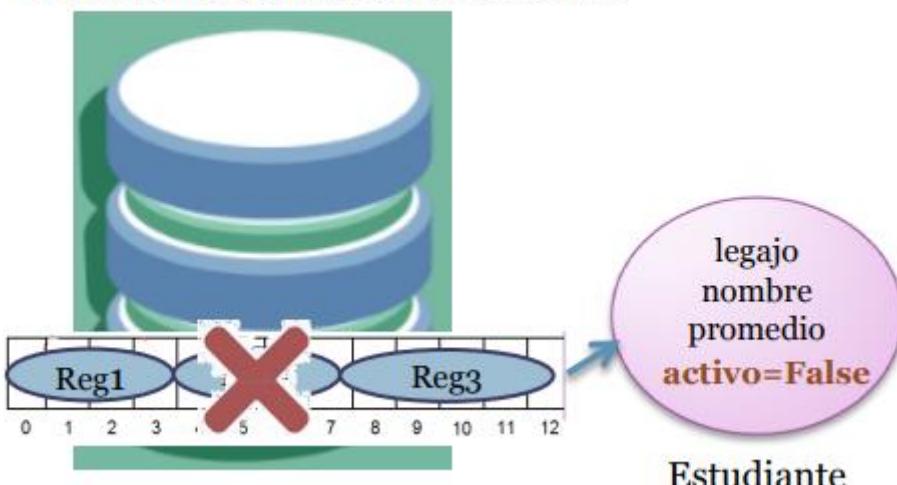
- Mecánica para la “baja” lógica en un archivo

1. Se marca usando un campo especialmente definido a modo de bandera.
2. Si este campo está marcado como “False” significa que el registro está marcado como

"eliminado". Pero si esta marcado como "True" significa que esta "activo".

3. Cuando se inserta en registro nuevo en el archivo se marca el campo en True.
4. Con un ciclo se recorre el archivo (considerando los registros activos), verificando si el campo clave coincide con el valor buscado. Cuando lo encuentra cambia el campo de marcado a False.

1) Archivo Original (**estudiantes.dat**)



- 2) Eliminar el estudiante=>correspondiente al "registro 2."

Paso a: Para dar de baja se debe ingresar el campo clave (legajo) del estudiante.

Paso b: Se recorre el archivo y en caso de encontrarlo se modifica el campo "activo" a False (baja lógica). La realización de esta última operación implica volver a grabar el registro.

Paso c: Cuando se procesa el archivo solo se considera los registros con campo "activo" en True

Estrategia óptima para la "baja" de registro

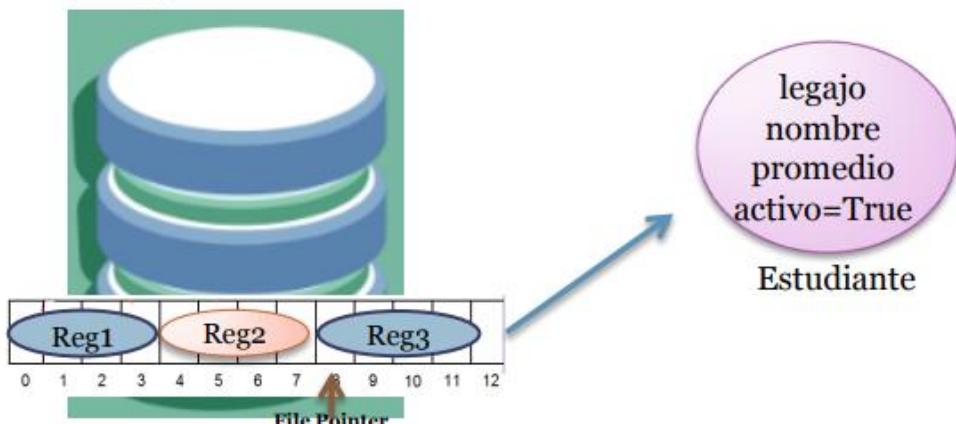
- Plantear una solución que equilibre las ventajas de ambas estrategias, se suele aplicar un "esquema combinado":
 - Baja Lógica: Para hacer bajas en tiempo real (por ejemplo, en casos de atención frente al público).
 - Baja Física: Se efectúa el proceso en algún momento no crítico de ciclo de uso del 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).

Gestión de ABM: Modificación de Registros

- Mecánica para el Modificación de un registro
 1. Se carga por teclado el valor del campo clave del registro (legajo del estudiante) a modificar.
 2. Con un ciclo se recorre el archivo y en cada iteración se consulta su clave (legajo del estudiante) que ingreso el usuario existe. En caso afirmativo se debe consultar si el campo activo es True.

3. Si se encuentra, se muestran los datos completos por pantalla, solicitando al usuario que campos a modificar. Carga por teclado los nuevos valores.
4. Para realizar la modificación se usa seek() para volver atrás el File Pointer y se vuelve a grabar el registro. (No se altera el campo activo, es decir que queda en True).

1) Archivo con 3 registros Alumnos ('estudiantes.dat')



2) Modificar "registro 2". Por ej: el promedio del estudiante.

- Paso a: Abrir archivo modo lectura y grabación ('r+b').
- Paso b: Ingresar el legajo del estudiante para su búsqueda y proceder a realizar la modificación.
- Paso c: Muestran los datos del estudiante encontrado.
- Paso d: Solicita al usuario que dato desea modificar del estudiante. En este caso por ejemplo se desea modificar el promedio. El usuario ingresa el nuevo promedio.
- Paso e: Se vuelve a grabar el registro con los cambios realizados

Cuestiones a tener en cuenta para la "modificación"

- En el proceso de serialización cuando se realiza un "alta" no todos los registros de un mismo tipo ocupan la misma cantidad de bytes.
- Esto implica un problema a la hora de realizar un proceso de gestión de ABM en un archivo, básicamente existe la probabilidad de que se puedan perder bytes de un determinado registro.
- Para solucionar este inconveniente la solución mas tradicional es mantener el mismo tamaño en bytes todos los registros.

Propuesta de solución para mantener el mismo tamaño de bytes en los registros de un archivo.

- En el caso de un campo tipo cadena el programador fija un criterio de longitud fija para el mismo.
- Ejemplo si se tiene el campo nombre con la cadena 'Luis' el programador fija que la longitud fija será de 30 caracteres.

```
r.nombre = 'Luis'
```

```
r.nombre = r.nombre.ljust(30,"")[:30]
```

Propuesta de solución para mantener el mismo tamaño de bytes en los registros de un archivo

- Si el campo es tipo numérico, el programador puede manejar este dato como una cadena. Y cuando sea necesario se realiza la conversión a número por medio de la función int(). Otra es validar que ese campo tenga un valor en un rango conocido (mismo tamaño en bytes).
- Si el registro tiene un campo cuyo valor es de tipo float entonces el programador no debe realizar ninguna acción, ya que los valores tipo float se representan con doble precisión (o sea 8 bytes por número).

Listados en archivos

- Los listados en archivos presentarse de dos formas:
 - Listado Completo: permite visualizar el contenido completo del archivo.
 - La mecánica es recorrer el archivo de manera secuencial y se visualizan todos los registros que no estén marcados como borrados.
- Listado Parcial: representa un listado con un determinado filtro.
- La mecánica es recorrer el archivo de manera secuencial, leyendo todos los registros del archivo, pero solo se visualizarán por consola aquellos que cumplen una determinada condición.

CUESTIONARIO

¿Cuáles de las siguientes son **ciertas** en relación a un *programa de gestión ABM*?
 (Aclaración: más de una respuesta puede ser válida. Marque **todas** las que considere correctas)

Seleccione una o más de una:

a.

Un programa de gestión ABM ofrece un menú para que el operador pueda realizar operaciones de altas, bajas y modificaciones sobre un archivo, pero normalmente ofrece además opciones para otras operaciones como listados completos o filtrados, búsqueda de registros por diversos criterios, ordenamiento, etc.

¡Correcto!

b.

Las operaciones básicas que un programa ABM permite realizar son las de altas, bajas y modificaciones de registros sobre un archivo.

¡Correcto!

c.

Típicamente, en un programa ABM, la opción del menú para realizar bajas lanza un proceso de baja por marcado lógico, y las verdaderas bajas físicas se hacen con un proceso separado de depuración del archivo (que no suele estar disponible en el menú del usuario final, sino que se activa en forma automática en momentos no críticos del funcionamiento del programa)

¡Correcto!



d.

Un programa de gestión ABM ofrece un menú para que el operador pueda realizar exclusiva y únicamente operaciones de altas, bajas y modificaciones sobre un archivo, y justamente de allí proviene la designación de ABM (iniciales de las palabras altas, bajas, modificaciones).

¿Cuál es la principal ventaja del *borrado físico* de un componente en un archivo?

Seleccione una:



a.

El borrado físico evita la necesidad de ordenar el archivo para una búsqueda rápida



b.

El borrado físico garantiza que luego sean posibles los recorridos secuenciales en el archivo.



c.

El borrado físico es más rápido que el marcado lógico



d.

El archivo no ocupa más lugar que el necesario, por lo que optimiza el espacio utilizado.

¡Correcto!

¿Cuál es la principal ventaja del *borrado por marcado lógico* en un archivo?

Seleccione una:



a.

El borrado por marcado lógico hace innecesario el borrado físico.



b.

El borrado por marcado lógico hace que el archivo ocupe sólo el espacio que necesita, optimizando el uso de la memoria externa.



c.

El borrado por marcado lógico es más rápido que el físico, siempre que la búsqueda del componente a eliminar pueda hacerse velozmente.

¡Correcto!



d.

El borrado por marcado lógico no requiere agregar atributos o campos especiales en los registros que se graban en el archivo.

Si se desea implementar una aplicación ABM sobre un archivos de registros, *con la intención de poder hacer seeking y ubicar el file pointer en forma directa en el byte donde comienza cada registro*, ¿cuál de las siguientes opciones indica los principales problemas que se deben enfrentar?

Seleccione una:



a.

El manejo de registros de tamaño uniforme grabados en el archivo.



b.

El manejo de registros que sólo contengan campos de tipo cadena de caracteres, pero todas con la misma cantidad de caracteres.



c.

El manejo de registros que sólo contengan campos numéricos de tipo flotante.



d.

El manejo de registros con campos de tipo cadena de caracteres de longitud variable, y en general, registros de tamaño no uniforme grabados en el archivo.

¡Correcto!

¿Cuál de las siguientes expone correctamente la estrategia general que debe llevar a cabo un *proceso de alta de registros en un archivo, suponiendo que NO se admiten registros con clave repetida* en ese archivo?

Seleccione una:



a.

1) Abrir el archivo *m* en modo 'a+b'. 2) Determinar si el archivo ya contiene o no a un registro con la misma clave que el registro *r* que se quiere agregar. 3) Si ya existe un registro con esa clave, rechazar el alta. Si no existe, asegurarse de marcar el registro como *eliminado* (*activo = False*), y grabar finalmente el registro en el archivo.



b.

1) Abrir el archivo *m* en modo 'ab'. 2) Determinar si el archivo ya contiene o no a un registro con la misma clave que el registro *r* que se quiere agregar. 3) Si ya existe un registro con esa clave, rechazar el alta. Si no existe, asegurarse de marcar el registro como *no eliminado* (*activo = True*), y grabar finalmente el registro en el archivo.



c.

1) Abrir el archivo *m* en modo 'a+b'. 2) Asegurarse de marcar el registro como *no eliminado* (*activo = True*), y grabar finalmente el registro en el archivo.



d.

1) Abrir el archivo *m* en modo 'a+b'. 2) Determinar si el archivo ya contiene o no a un registro con la misma clave que el registro *r* que se quiere agregar. 3) Si ya existe un registro con esa clave, rechazar el alta. Si no existe, asegurarse de marcar el registro como *no eliminado* (*activo = True*), y grabar finalmente el registro en el archivo.

¡Correcto!

¿Cuál de las siguientes expone correctamente la estrategia general que debe llevar a cabo un *proceso de baja lógica de registros en un archivo*?

Seleccione una:



a.

1) Abrir el archivo *m* en modo 'r+b'. 2) Determinar si el archivo contiene o no a un registro con la misma clave que se quiere eliminar. 3) Si no existe un registro con esa clave, rechazar la baja. Si existe, asegurarse de marcar el registro como *eliminado* (*activo = False*), y volver a grabar el registro en la misma posición que tenía en el archivo.

¡Correcto!



b.

1) Abrir el archivo *m* en modo 'rb'. 2) Determinar si el archivo contiene o no a un registro con la misma clave que se quiere eliminar. 3) Si no existe un registro con esa clave, rechazar la baja. Si existe, asegurarse de marcar el registro como *eliminado* (*activo = False*), y volver a grabar el registro en la misma posición que tenía en el archivo.



c.

1) Abrir el archivo *m* en modo 'a+b'. 2) Determinar si el archivo contiene o no a un registro con la misma clave que se quiere eliminar. 3) Si no existe un registro con esa clave, rechazar la baja. Si existe, asegurarse de marcar el registro como *eliminado* (*activo = False*), y volver a grabar el registro en la misma posición que tenía en el archivo.



d.

1) Abrir el archivo *m* en modo 'r+b'. 2) Determinar si el archivo contiene o no a un registro con la misma clave que se quiere eliminar. 3) Si no existe un registro con esa clave, rechazar la baja. Si existe, asegurarse de marcar el registro como *no eliminado* (*activo = True*), y volver a grabar el registro en la misma posición que tenía en el archivo.

¿Cuál de las siguientes expone correctamente la estrategia general que debe llevar a cabo un *proceso de modificación de registros en un archivo*?

Seleccione una:



a.

1) Abrir el archivo *m* en modo 'ab'. 2) Determinar si el archivo contiene o no a un registro con la clave que se está buscando. 3) Si no existe un registro con esa clave, rechazar la modificación. Si existe, ofrecer al operador un menú para modificar los campos que deseé. 4.) Asegurarse que el registro esté marcado como *no eliminado* (*activo = True*), y volver a grabar el registro en la misma posición que tenía en el archivo.



b.

1) Abrir el archivo *m* en modo 'a+b'. 2) Determinar si el archivo contiene o no a un registro con la clave que se está buscando. 3) Si no existe un registro con esa clave, rechazar la modificación. Si existe, ofrecer al operador un menú para modificar los campos que deseé. 4.) Asegurarse que el registro esté marcado como *eliminado* (*activo = False*), y volver a grabar el registro en la misma posición que tenía en el archivo.



c.

1) Abrir el archivo *m* en modo 'w+b'. 2) Determinar si el archivo contiene o no a un registro con la clave que se está buscando. 3) Si no existe un registro con esa clave, rechazar la modificación. Si existe, ofrecer al operador un menú para modificar los campos que deseé. 4.) Asegurarse que el registro esté marcado como *no eliminado* (*activo = True*), y volver a grabar el registro en la misma posición que tenía en el archivo.



d.

1) Abrir el archivo *m* en modo 'r+b'. 2) Determinar si el archivo contiene o no a un registro con la clave que se está buscando. 3) Si no existe un registro con esa clave, rechazar la modificación. Si existe, ofrecer al operador un menú para modificar los campos que deseé. 4.) Asegurarse que el registro esté marcado como *no*

eliminado (activo = True), y volver a grabar el registro en la misma posición que tenía en el archivo.

¿Cuál de las siguientes expone correctamente la estrategia general que debe llevar a cabo un *proceso de depuración física de un archivo*, para eliminar efectivamente en ese proceso a todos los registros marcados como eliminados en ese archivo?

Seleccione una:



a.

1) Abrir el archivo *original* y el *temporal* en modo 'wb'. 2) Leer uno a uno los registros del *original*, y grabar en el *temporal* sólo aquellos que NO estén marcados como eliminados. 3) Eliminar el archivo *original* y cambiar el nombre del *temporal*. 4) Cerrar ambos archivos.



b.

1) Abrir el archivo *original* y el *temporal* en modo 'wb'. 2) Leer uno a uno los registros del *original*, y grabar en el *temporal* sólo aquellos que estén marcados como eliminados. 3) Eliminar el archivo *original* y cambiar el nombre del *temporal*. 4) Cerrar ambos archivos.



c.

1) Abrir el archivo *original* y el *temporal* en modo 'wb'. 2) Leer uno a uno los registros del *original*, y grabar en el *temporal* sólo aquellos que NO estén marcados como eliminados. 3) Cerrar ambos archivos. 4) Eliminar el archivo *original* y cambiar el nombre del *temporal*.



d.

1) Abrir el archivo *original* en modo 'rb' y abrir el archivo *temporal* en modo 'wb'. 2) Leer uno a uno los registros del *original*, y grabar en el *temporal* sólo aquellos que NO estén marcados como eliminados. 3) Cerrar ambos archivos. 4) Eliminar el archivo *original* y cambiar el nombre del *temporal*.

¡Correcto!

Cuál de las siguientes expone correctamente la estrategia general que debe llevar a cabo un *proceso de listado completo de los registros de un archivo*, suponiendo que los registros *contienen un campo de marcado lógico* para gestionar eventuales bajas lógica?

Seleccione una:



a.

1) Abrir el archivo *m* en modo 'rb'. 2) Usar un ciclo para leer uno por uno los registros del archivo. 3) En cada vuelta del ciclo, si el registro leído está marcado como válido

(*activo = True*), mostrar su contenido; en caso contrario ignorarlo (no mostrar su contenido).

¡Correcto!



b.

1) Abrir el archivo *m* en modo 'rb'. 2) Usar un ciclo para leer uno por uno los registros del archivo. 3) En cada vuelta del ciclo, si el registro leído está marcado como *no válido* (*activo = False*), mostrar su contenido; en caso contrario ignorarlo (no mostrar su contenido).



c.

1) Abrir el archivo *m* en modo 'rb'. 2) Usar un ciclo para leer uno por uno los registros del archivo. 3) En cada vuelta del ciclo, si el registro leído está marcado como válido (*activo = True*), mostrar su contenido; en caso contrario no mostrar su contenido y detener el ciclo.



d.

1) Abrir el archivo *m* en modo 'ab'. 2) Usar un ciclo para leer uno por uno los registros del archivo. 3) En cada vuelta del ciclo, si el registro leído está marcado como válido (*activo = True*), mostrar su contenido; en caso contrario ignorarlo (no mostrar su contenido).

¿Cuál de las siguientes expone correctamente la estrategia general que debe llevar a cabo un *proceso de listado con filtro de los registros de un archivo*, suponiendo que los registros *contienen un campo de marcado lógico* para gestionar eventuales bajas lógica?

Seleccione una:



a.

1) Abrir el archivo *m* en modo 'rb'. 2) Usar un ciclo para leer uno por uno los registros del archivo. 3) En cada vuelta del ciclo, si el registro leído está marcado como *no válido* (*activo = False*) y además cumple la condición de filtro, mostrar su contenido; en caso contrario ignorarlo (no mostrar su contenido).



b.

1) Abrir el archivo *m* en modo 'rb'. 2) Usar un ciclo para leer uno por uno los registros del archivo. 3) En cada vuelta del ciclo, si el registro leído está marcado como válido (*activo = True*) y además cumple la condición de filtro, mostrar su contenido; en caso contrario ignorarlo (no mostrar su contenido).

¡Correcto!



c.

1) Abrir el archivo *m* en modo 'rb'. 2) Usar un ciclo para leer uno por uno los registros del archivo. 3) En cada vuelta del ciclo, si el registro leído está marcado como válido (*activo = True*) y además cumple la condición de filtro, mostrar su contenido; en caso contrario no mostrar su contenido y detener el ciclo.

d.

1) Abrir el archivo *m* en modo 'rb'. 2) Usar un ciclo para leer uno por uno los registros del archivo. 3) En cada vuelta del ciclo, si el registro leído está marcado como válido (*activo = True*), mostrar su contenido; en caso contrario ignorarlo (no mostrar su contenido).