**Temas semana 3 y 4;**

**Problemas al programar**

* Analizar un problema es tratar de entenderlo. Esta etapa busca garantizar que no tratemos de resolver un problema diferente al que tiene el cliente.
* Descomponer el problema en sus tres aspectos fundamentales, facilita la tarea de entenderlo: en cada etapa nos podemos concentrar en sólo uno de ellos, lo cual simplifica el trabajo.
* Esta descomposición se puede generalizar para estudiar todo tipo de problemas, no sólo se utiliza en problemas cuya solución sea un programa de computador.
* Además de entender el problema, debemos expresar lo que entendemos siguiendo algunas convenciones.
* Al terminar la etapa de análisis debemos generar un conjunto de documentos que contendrán nuestra comprensión del problema. Con dichos documentos podemos validar nuestro trabajo, presentándoselo al cliente y discutiendo con él.

**Buenas Prácticas en Programación:**

* Escribe tus programas lo más simple y directo posible. *Keep it simple*.
* Si trabajas con un compilador, ajusta sus opciones para que arroje la mayor cantidad de errores y advertencias posibles al compilar, de ese modo, tu aplicación tendrá menores chances de obtener errores aleatorios. En base a lo anterior, revisa cada mensaje para tomar las medidas pertinentes.
* Todo programa debe ser previamente comentado, explicando el propósito, funcionamiento completo y el resultado esperado.
* Dentro de las funciones definidas, establece un espaciado o **indentación**, que resalte la estructura funcional de la aplicación y facilite la lectura al programador al que le corresponda analizar el código.
* Por lo general, se usa un nivel de indentación por cada bloque de código (sentencias condicionales y bucles son consideradas como bloques de código embebido dentro de otro, por lo que se recomienda la indentación), ésta indentación corresponde a una sangría que comúnmente tiene el valor de una tabulación (tecla Tab) o bien tres o cuatro espacios.
* Es importante que el tamaño de las sangrías sean regulares (consistentes) y no varíen a lo largo del código, es decir, si el primer bloque ocupa como indentación una tabulación, el resto de bloques deben ser indentados con una tabulación adicional por cada nivel, con eso se facilita la lectura en cualquier editor de código.
* Se recomienda declarar variables en líneas separadas, ya que se facilita la descripción de cada variable mediante comentarios.
* Poner un espacio después de cada coma(,) facilita la legibilidad del código.
* No uses variables cuyo nombre no posea algún significado descriptivo, una variable con nombres significativos permite al lector entender el contexto del código y permite disminuir la cantidad de documentación asociada, puesto que con un código legible y nombres significativos, el código se ve *auto documentado*. Por ejemplo, una variable llamada cantidad\_recursos, tiene más significado que una variable llamada c.
* Se consistente al momento de utilizar un estándar para nombres largos, puedes usar el estándar usado en C ("nombre\_largo"), o bien el utilizado en Java, llamado CamelCase("nombre","VariableNombreLargo","Clase","ClaseNombreLargo").
* Evita el código *commented-out*, que corresponde al código comentado para que no se ejecute/no compile, ya que la lectura del código se vuelve engorrosa.
* Comenta cuando sea justo y necesario, usa los comentarios dentro de las funciones para describir las variables (sólo cuando su utilidad sea potencialmente dudosa) y cuando existan bloques de código difíciles de entender a primera vista; el exceso de comentarios vuelve ilegible el código.
* Se recomienda como buena costumbre, añadir al inicio de cada función, un bloque de comentarios que expliquen el comportamiento general de la función, de modo que se pueda entender a grosso modo que es lo que hace, o se espera que haga, así se facilita la búsqueda de errores, y se evita el análisis innecesario en una gran cantidad de casos.
* Es **altamente recomendada** la definición de variables locales al inicio de la implementación de cada función, como un bloque de código bien separado del bloque que contenga las instrucciones ejecutables, ésta separación puede consistir en una linea en blanco, o bien un comentario que denote la utilidad de cada bloque.
* En caso de usar operadores binarios (por ejemplo +, -, &&, ||, entre otros) se recomienda poner espacio a los extremos de cada operador, de modo que se resalte su presencia y se facilite la lectura del código.
* Se recomienda en algunas operaciones complejas, hacer uso de paréntesis redundantes o innecesarios que sirven para poder agrupar expresiones dentro de tales operaciones.
* **Evita la incorporación de más de una instrucción por línea.** Ésto reduce notoriamente la legibilidad del código, ya que el programador habitualmente está acostumbrado a leer una instrucción por línea.
* Si el código soporta la separación de sentencias en varias lineas, procura realizar una separación coherente, en el que cada punto de ruptura tenga sentido.
* Si una instrucción abarca más de una linea, recuerda realizar la indentación necesaria.
* Cuando escribas operaciones que hagan uso de muchos operadores, procura revisar que las operaciones se estén realizando en el orden que tu esperas que se realicen, muchas veces el lenguaje tiene otra forma de asimilar la precedencia, por lo que el resultado real varía con respecto al esperado, en general, se recomienda forzar la precedencia de operaciones haciendo uso de paréntesis.
* Si el lenguaje soporta llaves({}) para la separación de bloques, es altamente recomendado usarlas, ello facilita el proceso de distinción de bloques de código en forma rápida, permitiendo identificar y reparar errores en el código con menos dificultad.
* Si deseas evitar omitir una llave, abre y cierra el bloque de código que deseas crear, y luego introduce código dentro del bloque, con eso te aseguras la victoria.
* **Nunca** olvides inicializar los contadores y sumadores.
* 

**Identacion :**  Este término significa mover un bloque de texto hacia la derecha insertando espacios o tabuladores, para así separarlo del margen izquierdo y mejor distinguirlo del texto adyacente; en el ámbito de la imprenta, este concepto siempre se ha denominado *sangrado* o *sangría*.

En los lenguajes de programación de computadoras, la indentación es un tipo de notación secundaria utilizado para mejorar la legibilidad del código fuente por parte de los programadores, teniendo en cuenta que los compiladores o intérpretes raramente consideran los espacios en blanco entre las sentencias de un programa. Sin embargo, en ciertos lenguajes de programación como Haskell, Occam y Python, el sangrado se utiliza para delimitar la estructura del programa permitiendo establecer bloques de código.

Son frecuentes discusiones entre programadores sobre cómo o dónde usar el sangrado, si es mejor usar espacios en blanco o tabuladores, ya que cada programador tiene su propio estilo.

**Documentación:** La documentación para el usuario constituye un elemento de consulta para toda aquella persona que va a usar el programa por primera vez o que trata de saber si el programa servirá a sus objetivos. Igualmente es útil para usuarios que ya realizan un manejo básico y quieren profundizar hacia un conocimiento avanzado. Una documentación completa contendría:



· Portada con el nombre del programa, versión y autor o autores.

· Índice.

· Descripción muy breve de las funciones y posibilidades del programa.

· Descripción breve del método de cálculo principal.

· Explicación breve de cómo debe usarse el programa y de los datos de entrada, opciones y resultados.

· Ejemplos paso a paso de uso del programa en número suficiente para comprender las posibilidades que se brindan.

· Diagrama de flujo del programa de carácter sintético y descriptivo.

· Especificación detallada de todas las opciones contenidas en menús.

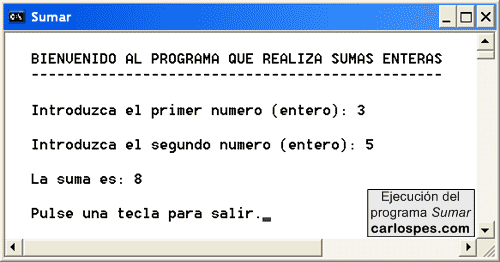
· Especificación detallada de todos los cálculos, principales y secundarios.

**Método de E/S de datos:**

Es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso. En este sentido; una biblioteca puede considerarse una base de datos compuesta en su mayoría por documentos y textos impresos en papel e indexados para su consulta. Actualmente, y debido al desarrollo tecnológico de campos como la informática y la electrónica, la mayoría de las bases de datos están en formato digital, siendo este un componente electrónico, por tanto se ha desarrollado y se ofrece un amplio rango de soluciones al problema del almacenamiento de datos.

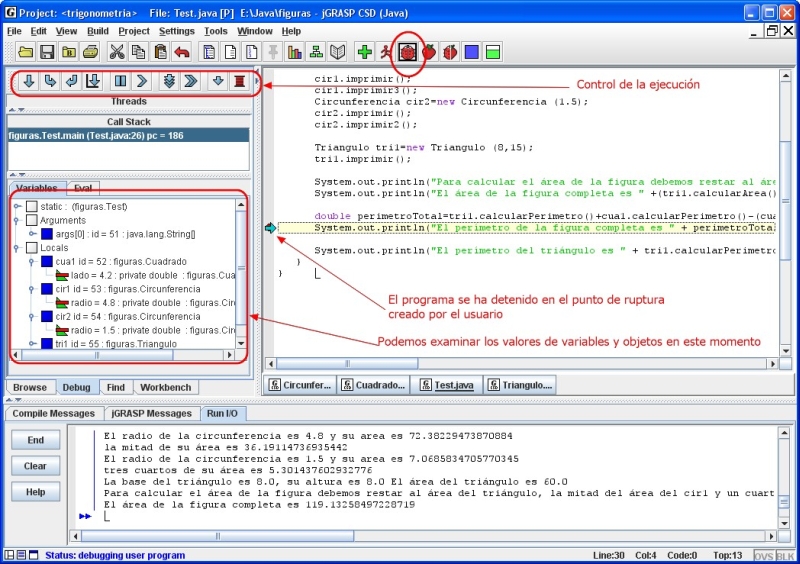
**Encoding:** El cifrado con fines de confidencialidad es un método utilizado para codificar una comunicación. Sólo el originador y el destinatario del mensaje puedan leerlo. Data shall be done **encoding** such data or using. such information is not accessible or manipulated during transport.

**Ejecución:** Ejecutar un programa implica que éste estará en estado de ejecución y, por ende, en memoria, hasta que se finalice.

  
  
Algunas bibliografías extienden la definición de ejecutar e incluyen la carga de archivos que no son ejecutables. Por ejemplo, cuando se ejecuta una imagen JPG y se abre el programa para visualizar la misma, o cuando se ejecuta un sonido MP3 y se abre el reproductor de audio. Técnicamente hablando, lo que está realmente en ejecución es el programa que se encarga de interpretar el archivo no ejecutable.

**Depuración:** es el proceso de identificar y corregir errores de programación. En inglés se conoce como *debugging*, porque se asemeja a la eliminación de *bichos* (*bugs*), manera en que se conoce informalmente a los errores de programación.

El término *bug* proviene de la época de las computadoras de válvula termoiónica, en los cuales los problemas se generaban por los insectos que eran atraídos por las luces y estropeaban el equipo. Si bien existen técnicas para la revisión sistemática del código fuente y se cuenta con medios computacionales para la detección de errores (depuradores) y facilidades integradas en los sistemas *lower CASE* y en los ambientes de desarrollo integrado, sigue siendo en buena medida una actividad manual, que desafía la paciencia, la imaginación y la intuición de programadores. Muchas veces se requiere incluir en el código fuente instrucciones auxiliares que permitan el seguimiento de la ejecución del programa, presentando los valores de variables y direcciones de memoria y ralentizando la salida de datos ("modo de depuración"). Dentro de un proceso formal de aseguramiento de la calidad, puede ser asimilado al concepto de "pre unitaria"

.

**Definición de clases:** C# es un lenguaje orientado a objetos puro[6], lo que significa que todo con lo que vamos a trabajar en este lenguaje son objetos. Un **objeto**es un agregado de datos y de métodos que permiten manipular dichos datos, y un programa en C# no es más que un conjunto de objetos que interaccionan unos con otros a través de sus métodos.

    Una **clase** es la definición de las características concretas de un determinado tipo de objetos. Es decir, de cuáles son los datos y los métodos de los que van a disponer todos los objetos de ese tipo. Por esta razón, se suele decir que el **tipo de dato** de un objeto es la clase que define las características del mismo[7].

**Sintaxis de definición de clases**

La sintaxis básica para definir una clase es la que a continuación se muestra:

|  |
| --- |
| **class** <nombreClase> {   <miembros> } |

**Clases públicas y Privadas:** Las clases permiten implementar tipos de datos abstractos. El problema que se presenta es que desde cualquier clase se puede accesar los campos de un objeto perteneciente a otra clase. Esto es una violación al principio de abstracción que dice que un tipo de datos abstracto sólo puede ser manipulado a través de las operaciones que se definen para éste.

#### Visibilidad de Campos y Métodos

En java al definir una clase se puede controlar la visibilidad que tendrán sus campos y métodos al exterior de la clase. Este control se efectúa de la siguiente forma:

class A

{

**private** int privx;

**protected** int protb;

**public** int pubc;

int paqd;

**private** void MetPriv() { ... }

**protected** void MetProt() { ... }

**public** void MetPub(A a)

{

// Un campo siempre es visible

// en la clase en donde se define

... privx ... a.privx ...

... protb ... a.protb ...

... pubc ... a.pubc ...

// Un método siempre es visible

// en la clase en donde se define

MetPriv(); a.MetPriv();

}

void MetPaq() { ... }

}

Las palabras **private**, **protected** y **public** son atributos de un campo o un método y su significado es el siguiente:

* **private**: El campo o método sólo es visible dentro de la clase donde se define.
* **protected**: El campo o método es visible en la clase en donde se define y en cualquiera de sus subclases.
* **public**: El campo o método es visible en cualquier clase.
* Ninguna de las anteriores: El campo o método es visible en cualquiera de las clases pertenecientes al paquete en donde se define.

**Atributos:** Los atributos son las características individuales que diferencian un objeto de otro y determinan su apariencia, estado u otras cualidades. Los atributos se guardan en variables denominadas de instancia, y cada objeto particular puede tener valores distintos para estas variables. Las variables de instancia también denominados miembros dato, son declaradas en la clase pero sus valores son fijados y cambiados en el objeto. Además de las variables de instancia hay variables de clase, las cuales se aplican a la clase y a todas sus instancias.

**Instancia:** En general, cuando se ejecuta un programa en un computador, se dice que éste se instancia. En los lenguajes de programación orientada a objetos un objeto es una instancia de una clase. Esto es, un miembro de una clase que tiene atributos en lugar de variables. En un contexto del mundo real, podríamos pensar en "Casa" como una clase y en un chalet como una instancia de esta e incluso otro chalet u otro tipo de casa como puede ser un apartamento como otra instancia.1​ En este caso no importa el tipo de casa, si fuese de nuestro interés modelarlo y especificarlo, diferenciaríamos entre un chalet y un apartamento con dos clases, entidades o prototipos diferentes, c.f. herencia (informática)

**Definición de funciones en programación:**

Definiendo Una **función** es un conjunto de líneas de código que realizan una tarea específica y puede retornar un valor. Las **funciones** pueden tomar parámetros que modifiquen su funcionamiento.

**Funciones Aritméticas:**

El signo más **(+)** se emplea para sumar dos valores, el signo menos **(-)** para restar un valor de otro, el asterisco **(\*)** para multiplicar dos valores, la división **(/)** para dividir un valor por otro, y el signo % para obtener el resto de una división entera. Estos símbolos se conocen como operadores binarios, pues operan sobre dos valores o variables.

La lista siguiente son ejemplos de expresiones aritméticas:

resultado = x - y;

total = capital+ interés;

cuadrado = x \* x;

celcius = (fahrenheit - 32) / 1.8

# Los operadores lógicos y sus tablas de verdad:

Aunque ya hay una amplia colección de artículos dedicados a Python, debemos admitir que hay bastantes elementos del lenguaje que aún no han sido presentados en familia. Es el caso, por ejemplo, de los **operadores lógicos**, de los que nos ocuparemos hoy. Mostraremos los distintos tipos y dejaremos que Python construya las respectivas **tablas de verdad**, ejemplo que nos servirá, a su vez, para ilustrar la **anidación de un bucle for dentro de otro**.

Con toda certeza que te has cruzado alguna vez en tu vida con los *operadores lógicos* o, como también se los conoce, *booleanos*. No tengo intención de darte una clase de *álgebra booleana*, de modo que nos centraremos directamente en su representación en Python.

Los tres operadores lógicos básicos son **O, Y y NO**, representados en Python por **or**, **and** y **not**, respectivamente.

Podemos incluir también el **O EXCLUSIVO**, que es verdadero cuando uno y solo uno de los operandos lo es, pero estrictamente debes saber que se deriva a partir de los tres básicos. Su representación es **^**, el sombrero o *caret*.

Como ejercicio práctico, vamos a realizar un programa que construya las *tablas de verdad* correspondientes.

booleanos = [False, True]

# Tabla de verdad de or

print('x\ty\tx or y')

print('-'\*22)

for x in booleanos:

    for y in booleanos:

        print(x, y, x or y, sep = '\t')

print()

# Tabla de verdad de and

print('x\ty\tx and y')

print('-'\*22)

for x in booleanos:

    for y in booleanos:

        print(x, y, x and y, sep = '\t')

print()

# Tabla de verdad de not

print('x\tnot x')

print('-'\*13)

for x in booleanos:

    print(x, not x, sep = '\t')

print()

# Tabla de verdad de ^

print('x\ty\tx ^ y')

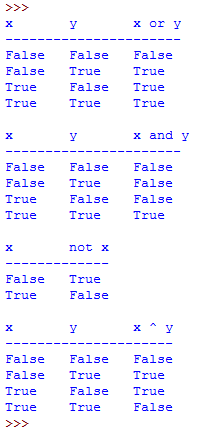
print('-'\*21)

for x in booleanos:

    for y in booleanos:

        print(x, y, x ^ y, sep = '\t')

No te preocupes si algo no te queda claro en el código, en breve lo explicaremos. Antes quiero que observes el resultado final:

[](http://elclubdelautodidacta.es/wp/wp-content/uploads/2012/11/python-logicos.png)

