16.5 Errores en Programación

Al programar es necesario revisar nuestro código por un compilador y los errores son inherentes al proceso de programación. Los errores de programación responden a diferentes tipos y pueden clasificarse dependiendo de la fase en que se presenten. Algunos tipos de errores son más difíciles de detectar y reparar que otros, veamos entonces:

- Errores de sintaxis
- Advertencias
- Errores de enlazado
- Errores de ejecución
- Errores de diseño

Errores de sintaxis son errores en el código fuente. Pueden deberse a palabras reservadas mal escritas, expresiones erróneas o incompletas, variables que no han sido declaradas, etc. Los errores de sintaxis se detectan en la fase de compilación. El compilador, además de generar el código objeto, nos dará una lista de errores de sintaxis. De hecho nos dará sólo una cosa o la otra, ya que si hay errores no es posible generar un código objeto.

Advertencias además de errores, el compilador puede dar también advertencias (Warnings). Las advertencias son errores, pero no lo suficientemente graves como para impedir la generación del código objeto. No obstante, es importante corregir estos errores la mayoría de las veces, ya que ante un aviso el compilador tiene que tomar decisiones, y estas no tienen por qué coincidir con lo que nosotros pretendemos hacer, ya se basan en las directivas que los creadores del compilador decidieron durante la creación del compilador. Por lo tanto en ocasiones, ignorar las advertencias puede ocasionar que nuestro programa arroje resultados inesperados o erróneos.

Errores de enlazado el programa enlazador también puede encontrar errores. Normalmente se refieren a funciones que no están definidas en ninguno de los ficheros objetos ni en las bibliotecas. Puede que hayamos olvidado incluir alguna biblioteca, o algún fichero objeto, o puede que hayamos olvidado definir alguna función o variable, o lo hayamos hecho mal.

Errores de ejecución incluso después de obtener un fichero ejecutable, es posible que se produzcan errores, durante la ejecución del código. En el caso de los errores de ejecución normalmente no obtendremos mensajes de error muy específicos o incluso puede que no obtengamos ningún error, sino que simplemente el programa terminará inesperadamente. Estos errores son más difíciles de detectar y corregir (pues se trata de la lógica como tal de nuestra aplicación). Existen herramientas auxiliares para buscar estos errores, son los llamados depuradores (Debuggers). Estos programas permiten detener la ejecución de nuestros programas, inspeccionar variables y ejecutar nuestro programa paso a paso (instrucción a instrucción). Esto resulta útil para detectar excepciones, errores sutiles, y fallos que se presentan dependiendo de circunstancias distintas. Generalmente los errores en tiempo de ejecución se dan por situaciones no consideradas en la aplicación, por ejemplo, que el usuario ingrese una letra en vez de un número y esto no se controle.

Errores de diseño finalmente los errores más difíciles de corregir y prevenir. Si nos hemos equivocado al diseñar nuestro algoritmo, no habrá ningún programa que nos pueda ayudar a corregirlos, pues es imposible que un programa pueda determinar qué es lo que tratamos de conseguir o un programa que realice aplicaciones cualquiera por nosotros. Contra estos errores sólo cabe practicar y pensar, realizar pruebas de escritorio, hacerle seguimiento y depuración a la aplicación hasta dar con el problema (una mala asignación, un valor inesperado, olvidar actualizar una variable, etc.), también es útil buscar un poco de ayuda de libros o en sitios y foros especializados.

En este resumen, se enumera algunas de las cosas que han aportado muchos programadores para todos aquellos que se inician en el arte y ciencia de la programación (recordando que se puede pasar más tiempo encontrando y corrigiendo errores que en el desarrollo de nuevas características en el código). Se leen rápido, pero aplicarlos bien puede llevar toda la vida.

1. Tome el código y divida grandes partes de código en pequeñas funciones.
2. Si para cuando sale del trabajo no ha resuelto el problema. Apague la computadora y déjela para el día siguiente. No pienses más en el problema.
3. Principio YAGNI: "No lo necesitará" o no codifique más de lo que se le pidió. No anticipe el futuro y simplemente cree algo que funcione

lo antes posible. Codifique solo las partes necesarias para resolver el problema actual.

4. No es necesario saberlo todo, ni todos los marcos existentes. Lo más significativo es tener una buena base. Conozca el lenguaje en profundidad antes de comenzar con un Framework y aprenda cosas fundamentales como los principios SOLID o cómo escribir código limpio.

5. KISS: "Keep it simple, stupid" o "keep it stupid simple" es un principio de diseño que establece que la mayoría de los sistemas funcionan mejor si se mantienen simples en lugar de complicados. Y si bien esto es lógico, a veces es difícil de lograr.

6. No lo pienses demasiado.

7. Si tiene un problema o un error durante demasiado tiempo, aléjese y vuelva más tarde. A menudo, las mejores soluciones a los problemas se me ocurren en el camino a la oficina, al baño. También es aconsejable alejarse cuando está enojado con un cliente o con un compañero de trabajo, especialmente si deseas conservar tu trabajo.

8. Aprenda a escribir pruebas útiles y aprenda a hacer TDD. TDD es un proceso de desarrollo de Software que se basa en la repetición de un ciclo de desarrollo muy corto: escriba una prueba, ejecute todas las pruebas y vea si la nueva falla, escriba algún código, ejecute pruebas, refactorice el código, repita.

9. Primero resuelva el problema y luego escriba el código. No empiece a codificar sin saber qué hacer.

10. No memorice el código, en su lugar, comprenda la lógica.

11. Si copia y pega una solución de desbordamiento de pila, asegúrese de comprenderla. Aprenda a usar Stack Overflow de una buena manera.

12. Si quieres aprender algo, practica. Haz ejemplos y haz que funcionen porque leer sobre algo no es suficiente.

13. Estudie el código de otras personas y deje que otros estudien su código de vez en cuando. La programación en pareja y las revisiones de código son una buena idea.

14. No reinvente la rueda.
15. Tu código es la mejor documentación.
16. Sepa cómo googlear cosas. Para ello, es necesario tener experiencia y leer mucho para saber qué buscar.
17. Tu código deberá ser mantenido por ti mismo en el futuro o por otros, así que escribe el código con el lector en mente, no tratando de ser la persona más inteligente. Haz que se lea como si estuvieras leyendo una historia.
18. La mejor manera de resolver un error con Google es copiarlo y pegarlo.
19. Nunca te rindas, al final, de una forma u otra lo resolverás. Hay días malos, pero pasarán.
20. Descanso, descanso y descanso. La mejor manera de resolver un problema es tener una mente tranquila.
21. Aprenda a utilizar los patrones de diseño de Software. Los patrones de diseño son soluciones a problemas comunes en el diseño de Software. Cada patrón es como un plano que puede personalizar para resolver un problema de diseño común en su código. (No reinvente la rueda)
22. Utilice herramientas de integración y automatice tanto como pueda.
23. Hacer katas de código. Un kata de código es un ejercicio de programación que ayuda a los programadores a mejorar sus habilidades mediante la práctica y la repetición.
24. Programe en una interfaz, no en una implementación. La inyección de dependencia es un requisito. Consulte los principios SOLID.
25. Refactor -Test-Refactor. La refactorización es una técnica para reestructurar un código existente, alterando y mejorando su estructura interna sin cambiar su comportamiento externo.
26. Pide ayuda cuando la necesites. No pierdas el tiempo.
27. La práctica hace la perfección.

28. Aunque a veces los comentarios pueden ayudarte, no les prestes demasiada atención. Probablemente estén desactualizados.

29. Conozca su entorno de desarrollo e invierta en uno lo suficientemente potente.

30. Reutilizar componentes.

31. Al desarrollar una aplicación Web, piense primero en los dispositivos móviles y en las restricciones de ancho de banda y energía asociadas.

32. No optimice ni refactorice antes de tiempo. Es más importante tener un producto mínimo viable lo antes posible.

33. Nunca elija una forma de atajo ineficiente para ahorrar unos minutos. ¡Cada vez que codifique, dé lo mejor!

34. Siga los estándares documentados.

35. Los usuarios no son personas técnicas. Piense en ello cuando desarrolle su interfaz de usuario.

36. Utilice siempre un sistema de control de código fuente como GitLab, Github o bitbucket y realice confirmaciones de git pequeñas y frecuentes.

37. Es mejor usar registros que depurar el código. Registre todas las partes críticas.

38. Sea consistente al codificar. Si usa un estilo, use siempre el mismo. Si trabaja con más personas, use el mismo estilo con todo el equipo.

39. No dejes de aprender, pero más que nuevos lenguajes o marcos, céntrate en los conceptos básicos del desarrollo de Software.

40. Y finalmente, paciencia y ama lo que haces.

16.5.1 Errores en Java

Ejemplo en Java:

```
class hola {
   public static void main(String[] args) {
      System.out.println("Hello world!");
   }
}
```

para compilarlo usamos *javac* en línea de comandos mediante:

```
$ javac hola.java
```

y lo ejecutamos con:

```
$ java hola
```

Errores en tiempo de compilación Los errores de compilación ocurren porque la sintaxis del lenguaje no es correcta, este tipo de errores no permiten que la aplicación se ejecute, por ejemplo:

- Olvidarnos de un punto y coma al final de una sentencia.
- No cerrar llaves en algún bloque de código, método, clase o en alguna estructura de control.
- Repetir variables con el mismo nombre aunque sean de diferente tipo.
- No distinguir bien los diferentes tipos primitivos (boolean, byte, short, int, long, float y double) y no saber elegir el más adecuado a cada problema.
- Asignar en una variable otra variable con tipo de dato diferente: es decir si tengo una variable String y su valor lo asigno en una variable de tipo int, en este caso el compilador me va dar un error de conversión de tipos y esto no va dejar que la aplicación compile, incluso un error parecido se puede dar en variables de grupos del mismo tipo, por ejemplo asignar el valor de un tipo int en un tipo short.

- Error en mayúsculas y minúsculas, Java es un lenguaje case sensitive, esto es que debemos respetar las mayúsculas y minúsculas. (miVar no es igual a mivar). Si hacemos referencia a un identificador que no está bien escrito entonces obtendremos un error de sintaxis.

- No llevar una cuenta clara y ordenada de los paréntesis (), llaves {} y comillas "" abiertos, lo cual conduce a que algunas ovejas vayan con parejas que no son las suyas o queden del todo desparejadas.

- No tener en cuenta el orden en que se realizan las operaciones en las expresiones de Java, de modo que en la práctica se obtiene un resultado distinto del que se espera.

- No poner bien las llaves en las sentencias if, while y for. Olvidarse por ejemplo de que una sentencia for(int i=0; i<n; i++); repite n veces la sentencia vacía (por la presencia del punto y coma final) en lugar de la sentencia siguiente al for.

- Usando equals contra la asignación (== versus =), en Java este error puede ser incluso un error de lógica o sintaxis. Para comparar 2 referencias para igualarlas se usa el operador == (el operador de igualdad). Para asignar el valor de la derecha a la variable de la izquierda se usa el operador = (operador de asignación). Los programadores novatos a veces escriben: if (miValor = valorEsperado) Este código intenta evaluar el valorEsperado como un valor booleano en lugar de intentar la evaluación de igualdad entre miValor y valorEsperado. Si valorEsperado es del tipo booleano, entonces el código tendrá un error de lógica y probará si el valorEsperado es verdadero o falso. Si valorEsperado no es del tipo booleano, entonces el código lanzará un error de compilación debido a que la estructura if requiere un valor booleano que sea retornado de la comparación (miValor = valorEsperado), pero en Java el operador = siempre retorna el valor de la derecha.

- No tener la idea clara de cómo funcionan los métodos, de lo que son los argumentos y de cómo se pasan, y de lo que es el valor de retorno y cómo se devuelve.

- Definir en un método una variable local con el mismo nombre que una variable miembro de su clase, y olvidarse de poner la referencia this cuando se quiere acceder a la variable miembro.

- Crear la referencia a un objeto y olvidarse de crear el objeto. Se obtiene un error cuando se intenta acceder a alguna variable miembro o método del objeto.
- No entender bien los métodos gráficos fundamentales (paint(), update() y repaint()) y no saber redefinirlos de acuerdo con el problema que se desea resolver.
- No saber encontrar en la documentación de Java lo que se necesita para resolver una determinada dificultad o problema de programación.
- No saber interpretar los mensajes de error del compilador (o no intentarlo siquiera), para obtener pistas que ayuden a su detección y corrección.
- No saber qué hacer cuando el problema compila y arranca bien, pero da errores en tiempo de ejecución o simplemente da un resultado incorrecto. "¿El debugger? Sí, me suena que era para algo de esto..."
- No tener una idea clara de lo que es un cast y de cuándo es imprescindible utilizarlo para que un programa funcione.
- No saber qué hacer cuando se desea utilizar un método de Java que lanza una excepción y nos da error al compilar por no haber capturado y gestionado dicha excepción.
- Tener un proyecto activo con una enorme confusión de ficheros y directorios, por ejemplo de modo que se utilicen y modifiquen ficheros del proyecto anterior destruyendo lo que ya funcionaba, etc., etc.
- No guardar con frecuencia los ficheros del proyecto, arriesgándose a perder en cualquier momento todo el trabajo realizado.
- No programar de una manera limpia, clara y ordenada; no introducir ningún comentario; elegir nombres de variables que no mantienen ninguna semejanza o relación con la magnitud a la que representan.
- No detenerse a pensar -preferiblemente sobre papel- el esquema general del programa a realizar. No intentar definir primero un esqueleto y luego todos los detalles de terminación. Conceder a todos los puntos exactamente la misma importancia.

- No desarrollar progresivamente, comprobando cada paso antes de pasar al siguiente. Olvidarse de que más vale un programa incompleto que lo que hace lo hace bien, que un programa que tiene todo pero que todo o casi todo lo hace mal.
- El nombre de la Clase pública no coincide con el nombre del archivo, cada archivo .java puede contener sólo una Clase pública. El nombre de esa Clase pública debe coincidir exactamente con el nombre del archivo antes de la extensión .java, respetando incluso las mayúsculas y minúsculas. Por ejemplo, una Clase pública llamada MyClass debe estar en un archivo MyClass.java y no en myclass.java. Este es un error de sintaxis.
- Una Clase no está en el directorio correcto, este error de sintaxis ocurre si el comando javac no puede encontrar un archivo .java en el directorio esperado. Si una Clase está en el paquete por defecto (no tiene la declaración del package en la primera línea), entonces esta Clase pertenece al directorio actual de trabajo o al directorio donde apunta el flag sourcepath del javac.

Errores en tiempo de ejecución Estos errores ocurren cuando la aplicación se está ejecutando, imagínate que cuando estás haciendo la presentación o probando tu aplicación en público o frente a tu profesor y de repente pummm la aplicación explota (se cuelga) por lo general a este tipo de errores se los conoce como errores de compilación, ahora, por qué ocurren estos errores?, bueno, hay muchos factores desde los más básicos como por ejemplo:

- El usuario ingresa valores diferentes a los que la aplicación recibe
- Acceder a una posición en un arreglo la cual no existe.
- Almacenar cadenas donde se debe almacenar números
- Divisiones por cero.
- Digamos que en una aplicación móvil consuma datos de un servicio web y que al momento de consumir esos datos no haya conexión a internet, esto hace que la aplicación se cuelgue.

- Olvidar que los índices en Java empiezan en 0, los índices de los arreglos de Java y las listas empiezan en 0, myArray[0], o myList.get(0). Asegurarse que su loop for no cause errores por este motivo. Si hacemos más loops de los que son posibles, entonces obtendremos el error: ArrayIndexOutBounds exception. Si hacemos menos loops de los requeridos, entonces tendremos un error de lógica.

- NullPointerException, el NullPointerException, la maldición de los programadores en Java. Los NullPointerExceptions son errores de lógica causados cuando un programa intenta acceder a métodos o atributos en una referencia que está nula. Si nuestro objeto no ha sido inicializado o ha sido establecido a null con el operador =, entonces la llamada el método o el acceso a uno de sus atributos no es válido.

Errores lógicos Estos son los más difíciles de detectar y corregir y pues bueno porque digo que son difíciles, es que con este tipo de errores la aplicación compila y se ejecuta de forma normal, pero, y entonces dónde está el error? El error se da porque la aplicación no muestra los resultados esperados digamos que ya hice el algoritmo y el resultado que debo obtener es 10, pero en la aplicación obtengo 5 o lo peor de todo que de seguro te a pasado, que escribes el código haces pruebas y los resultados son correctos subes a producción y resulta que para un caso particular no se obtuvo el resultado esperado. Pues en estos casos el problema son errores lógicos que a fin de cuentas estos ocurren por el mal diseño del algoritmo de la aplicación

Para terminar también mencionar que otro error de tipo lógico también ocurre cuando colocamos punto y coma después de una sentencia if o for y aunque parezcan errores tan obvios, se dan cuando estamos empezando a programar.

Por ello, estos son algunos consejos del buen programador:

- Tener muy cuenta las divisiones por cero, en cuanto a los arreglos no acceder a posiciones que no existen.

- Hacer uso de excepciones y validaciones cuando se tenga en cuenta que el usuario puede ingresar valores diferentes a los que se puede recibir la aplicación.

- Cuando diseñemos un algoritmo tener en cuenta todos los escenarios posibles que pueda tomar la aplicación durante su ejecución.

- Finalmente como dice el dicho divide y vencerás: una buena forma de contrarrestar los errores de tipo lógico es dividir algoritmos grandes y complejos en tareas pequeñas de forma que el código se más legible y cualquier error de tipo lógico se lo pueda depurar fácilmente.

Lista de errores comunes

- Incompatyble types: unexpected return value:

  (Tipos incompatibles: valor de retorno no esperado) El mensaje de error te indica que existen tipos incompatibles. ¿Qué hacer? Revisa el tipo de método y la instrucción return si la incluye.

  Recuerda que:

  Declaración del método void →Sentencia return no debe de existir en este tipo de método.

  Declaración del método int, float, double, char, boolean→Sentencia return debe existir, regresando un valor del tipo apropiado

- Cannot find symbol

  (No se puede encontrar el símbolo) El mensaje de error te indica que existe un símbolo que no se reconoce por que no está declarado. ¿Qué hacer? Revisa que la variable esté declarada como variable local del método, verifica que el nombre del método invocado sea el correcto, o asegúrate que el nombre de la variable global o local esté bien escrita.

- Invalid method declaration; return type required

  (Declaración inválida del método; tipo de regreso requerido) El mensaje de error te indica el método no ha sido declarado apropiadamente y que se requiere un tipo. ¿Qué hacer? Revisa que el método tenga especificado un tipo de método, ya sea alguno de los tipos primitivos de datos o incluso en tipo void si no regresará valores, la declaración le falta el tipo de valor que el método regresará, ya sea int, float, char, double, boolean o void.

- Ilegal start of expression

(Comienzo ilegal de expresión) Este mensaje de error aparece al inicio de un nuevo método, ya sea uno definido por el usuario o el mismo main. Puedes pasar horas revisando la sentencia donde el error se posiciona, y encontrarás que no hay error. Y eso es debido a que un método anterior no se ha cerrado, cualquier inicio de otro método generará error de sintaxis. ¿Qué hacer? Revisa que el método anterior tenga su llave de cierre.

- Missing return statement

  (Falta la sentencia return) Este error se genera simplemente porque se ha olvidado la sentencia return, la cual es obligatoria en los métodos de tipo int, char, float, double y boolean; además de cualquier tipo definido por el usuario. ¿Qué hacer? Asegúrate que, si el método no es de tipo void, exista la sentencia de return.

- class, interface or enum expected

  (Se esperabla class, interface o enum) El mensaje explica que se espera el inicio de una clase, interface o enumeración en esa parte del código. ¿Por qué? Muy sencillo, porque la clase actual ya ha sido cerrada. ¿Qué hacer? Revisa que en el método anterior no sobre una llave de cierre.

Esta es una lista muy pequeña de los errores más comunes que se pueden presentar cuando se trabaja con un programa modular.

16.5.2 Errores en Pyhton

Python es un lenguaje potente y flexible con muchos mecanismos y paradigmas que pueden mejorar significativamente el rendimiento. Sin embargo, como con cualquier herramienta de software o lenguaje, con una comprensión o evaluación limitada de sus capacidades, pueden surgir problemas imprevistos durante el desarrollo.

Ejemplo en Python 2:

```
print "Hello World\n"
```

para compilarlo y ejecutarlo usamos *python2* en línea de comandos mediante:

```
$ python2 hola.py
```

Ejemplo en Python 3:

```
print ("Hello World\n")
```

para compilarlo y ejecutarlo usamos *python3* en línea de comandos mediante:

```
$ python3 hola.py
```

Errores en tiempo de compilación Los errores de compilación ocurren porque la sintaxis del lenguaje no es correcta, este tipo de errores no permiten que la aplicación se ejecute, por ejemplo:

- Python distingue entre las mayúsculas y las minúsculas ("case sensitive").

  MapSize = "34x44"

  mapsize = "22x34"

- Indentación incorrecta

  Atención a la alineación del código. Esto es, a la hora de escribir funciones, etc., existen una sangrías que deben respetarse para que no nos de error. Mientras que la mayoría de los lenguajes utilizan la sangría para mejorar su legibilidad, pero no dependen de la práctica, Python ha tejido la sangría directamente en el tejido de su lenguaje. Esto significa que no puede permitirse cometer errores cuando se trata de formatear su código. Para Python, la regla es tener 4 espacios para sangrar. No mezcles con 2,3,5 o una cantidad diferente de espacios ya que Python simplemente no ejecutará tu programa. Necesitarás aprender a usar Python para separar tus bloques de código para que todo fluya bien. Es importante asegurarse de que lo haga bien la primera vez. No sangrar correctamente puede llevar a un error en su código si no tiene cuidado. Debugging ya es bastante pesado por sí solo, pero buscar un espacio que falta o un espacio extra te volverá loco. Por

ejemplo en el caso de la sentencia "except" debe estar a la altura de "try" ya que funcionan como un bloque. Todas las sentencias tabuladas a la misma distancia pertenecen al mismo bloque de código hasta que no se encuentre una línea con menor tabulación

- Hay que tener cuidado a la hora de escribir las rutas para encontrar tus datos. La barra que entiende Python es / ó \\ (en esta posición \ posee otro significado para este lenguaje de programación).
- Los espacios como guión bajo. Hay ocasiones que escribimos las carpetas con espacios o recibimos datos que los incluye, es una mala costumbre al trabajar. A la hora de escribir tenemos que tenerlo muy en cuenta y poner dicho guión bajo.
- Al realizar una comparación entre dos objetos o valor, se tiene que utilizar el operador de igualdad (==), no el operador de asignación (=). El operador de asignación coloca un objeto o valor dentro de una variable y no compara nada.
- Olvidar poner los ":"al final de las sentencias tipo: try, if, elif, else, for, while, class, o def.
- Identificados con el código SyntaxError, son los que podemos apreciar repasando el código, por ejemplo al dejarnos de cerrar un paréntesis:

  ```
  print("Hola"
  ```

- Errores de nombre, se producen cuando el sistema interpreta que debe ejecutar alguna función, método... pero no lo encuentra definido. Devuelven el código NameError:

  ```
  pint("Hola")
  ```

La mayoría de errores sintácticos y de nombre los identifican los editores de código antes de la ejecución, pero existen otros tipos que pasan más desapercibidos.

Errores semánticos, Estos errores son muy difíciles de identificar porque van ligados al sentido del funcionamiento y dependen de la situación. Algunas veces pueden ocurrir y otras no. La mejor forma de prevenirlos es programando mucho y aprendiendo de tus propios fallos, la experiencia es la clave. Veamos un par de ejemplos:

- Ejemplo pop() con lista vacía

  Si intentamos sacar un elemento de una lista vacía, algo que no tiene mucho sentido, el programa dará fallo de tipo IndexError. Esta situación ocurre sólo durante la ejecución del programa, por lo que los editores no lo detectarán:

  ```
  l = []
  l.pop()
  ```

  Para prevenir el error deberíamos comprobar que una lista tenga como mínimo un elemento antes de intentar sacarlo, algo factible utilizando la función len():

  ```
  l = []
  if len(l) > 0:
      l.pop()
  ```

- Ejemplo lectura de cadena y operación sin conversión a número

  Cuando leemos un valor con la función input(), éste siempre se obtendrá como una cadena de caracteres. Si intentamos operarlo directamente con otros números tendremos un fallo TypeError que tampoco detectan los editores de código:

  ```
  n = input("Introduce un número: ")
  print("{}/{} = {}".format(n,m,n/m))
  ```

  Como ya sabemos este error se puede prevenir transformando la cadena a entero o flotante:

  ```
  n = float(input("Introduce un número: "))
  m = 4
  print("{}/{} = {}".format(n,m,n/m))
  ```

Sin embargo no siempre se puede prevenir, como cuando se introduce una cadena que no es un número:

```
n = float(input("Introduce un número: "))
m = 4
print("{}/{} = {}".format(n,m,n/m))
```

Como podemos suponer, es difícil prevenir fallos que ni siquiera nos habíamos planteado que podían existir. Por suerte para esas situaciones existen las excepciones.