Introducción a la Programación

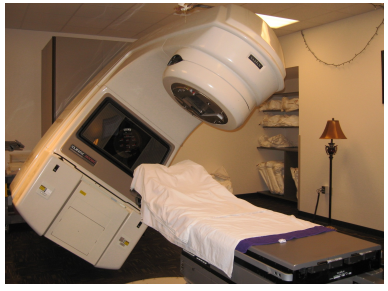
Prof. Agustín Gravano

Primer semestre de 2022

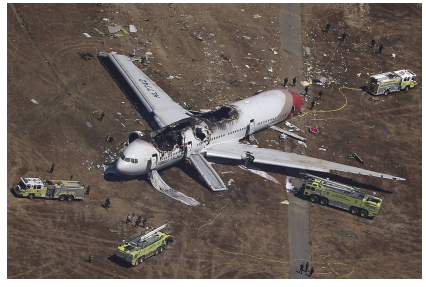
Clase teórica 6: Errores y testing

Errores de software

Los errores de software pueden causar enormes pérdidas de dinero y vidas.



Therac-25, 3 muertos (1985)

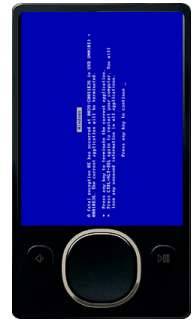


Airbus A300, 264 muertos (1994)

Knight Capital, 440M USD (2012)

Ariane 5 (1996)

Y2K (1999/2000)



MS Zune (2008/2009)

http://www.cse.psu.edu/~gxt29/bug/softwarebug.html

Verificación formal de programas

En teoría, es posible demostrar (como un teorema matemático) que un programa es correcto respecto de una especificación formal, escrita por ejemplo en lógica de primer orden:

Encabezado:	`división_entera`: $a \in \mathbb{Z} \times b \in \mathbb{Z} \to \mathbb{Z}$
Requiere:	$b \neq 0$
Devuelve:	$\mathrm{q} \in \mathbb{Z}$ tal que $(\exists r \in \mathbb{Z})\ 0 \leq r < \lvert b \rvert \wedge a = \mathrm{q} * b + r$

Pero demostrar la correctitud manualmente es difícil, laborioso y (también) propenso a errores.

Hay mucho trabajo de investigación en la creación de herramientas (¡programas!) capaces de verificar automáticamente pequeñas porciones de código crítico: *model checkers*, demostradores de teoremas, etc.

Pero (siempre hay un pero...) esto también es difícil y trabajoso, y no siempre es posible. Suele usarse muy excepcionalmente.

Calidad del software

Por eso, la calidad de nuestros programas termina dependiendo de:

1. Buenas prácticas de programación
2. Testing
3. Verificación entre pares:
   - Programar en equipos de 2-3 personas.
   - Forzar la inspección y aprobación del código por parte de otros programadores antes de subirlo a producción.
   - Muchos ojos ven más que dos.
   - (Trabajos Prácticos de las materias de programación)

Errores de software

En un programa pueden suceder varios tipos de error:

1. Errores de sintaxis
2. Errores en tiempo de ejecución
3. Errores semánticos

Es útil distinguirlos y entenderlos, para poder encontrarlos y arreglarlos rápidamente.

1. Errores de sintaxis

Ocurren cuando nuestro código no respeta las reglas sintácticas del lenguaje de programación.

Ejemplos frecuentes:

- Nombrar a una variable con una palabra reservada (ej: `float`, `def`, `import`, `return`, etc.).
- Escribir expresiones con paréntesis, comillas, etc. desbalanceados:
  - `((1+2)/2`
  - `'hola mundo"`
- Usar indentación inconsistente:

```
1  def f(x:int) -> int:
2      x = x + 1
3    return x
```

- Usar = (asignación) cuando la intención era usar == (comparación).
- ...

1. Errores de sintaxis

El intérprete de Python suele ayudarnos a encontrar el error.

Los mensajes de error incluyen detalles útiles, a los cuales conviene prestar atención.

Ejemplos:

```
[...]

  File "/home/agravano/tmp/untitled01.py", line 32
    def = 24
        ^
SyntaxError: invalid syntax
```

```
[...]

  File "/home/agravano/tmp/untitled02.py", line 14
    return x
           ^
IndentationError: unindent does not match any outer
    indentation level
```

2. Errores en tiempo de ejecución

En este caso, el programa es sintácticamente correcto, y el intérprete de Python puede comenzar a ejecutarlo, hasta que se encuentra con un problema.

Ejemplos frecuentes:

- Error de nombres (`NameError`): cuando se intenta usar una variable o una función que aún no fue definida.
- Error de tipos (`TypeError`): cuando se intenta usar un valor de manera inadecuada. Ejemplo: `a+1`, si a vale `'xx'`.
- Error de índices (`IndexError`): cuando se intenta acceder a una posición inexistente en un string (o una lista, como veremos más adelante). Ejemplo: `a[3]`, si a vale `'xyz'`.
- División por cero (`ZeroDivisionError`).
- El programa no termina nunca, por ejemplo por un ciclo infinito. (Sobre este punto volveremos dentro de un par de clases.)
- ...

2. Errores en tiempo de ejecución

Para encontrar y corregir estos errores, es necesario tener control de la evolución del estado del programa.

Para cada momento de la ejecución, deberíamos poder determinar, para cada variable:

- Si está definida.
- Su tipo. (Ayuda fijar el tipo de cada variable con *type hints*.)
- Su valor actual.

Una herramienta útil para conocer el valor de las variables es incluir `print`s alrededor del lugar donde sospechamos que se produce el error.

Otra estrategia es usar un *debugger* (por ejemplo, el incluido con el IDE Spyder), que permite controlar paso a paso la ejecución del programa e inspeccionar sus variables.

3. Errores semánticos

En este caso, el programa se ejecuta sin inconvenientes, pero su comportamiento no es el esperado: hace algo distinto a lo que queremos.

Los errores semánticos son difíciles de detectar: solo nosotros sabemos qué debería hacer un programa, en contraste con lo que realmente hace.

Aquí entran otra vez en juego los ejemplos de uso que construimos al especificar las funciones. Con ellos ejecutaremos testing de unidad para poner a prueba las unidades modulares de nuestro código (ej.: funciones).

Para cada entrada definida en un caso de test, revisaremos (en forma automática) que produzca la salida esperada.

Importante: Esto apunta a ganar confianza en un programa, pero **nunca** (literalmente) tendremos la certeza absoluta de que no contiene errores.

> El testing puede usarse para mostrar la presencia de errores; pero nunca para demostrar su ausencia. (Edsger W. Dijkstra)

Testing de unidad · Ejemplo

La clase pasada especificamos la función:

$$\texttt{cant\_vocales(texto:str)} \to \texttt{int}$$

y construimos este conjunto de ejemplos de uso:

texto	**Resultado**	Criterio/comentarios
`''`	0	caso vacío
`'aáAÁ'`	4	variantes de vocales
`'aeiouAEIOUáéíóúüÁÉÍÓÚÜ'`	22	variantes de vocales
`'bcdfghjklmnñpqrstvwxyz'`	0	consonantes minúsculas
`'BCDFGHJKLMNÑPQRSTVWXYZ'`	0	consonantes mayúsculas
`',.;:¿?¡!-\'" '`	0	signos puntuación/espacios en blanco
`'eeeee'`	5	vocales repetidas

Vamos a armar un testing de unidad en Python usando estos ejemplos.

Es una buena práctica hacer esto antes de implementar la función.
Mejor aún si lo hacen personas distintas.

Testing de unidad en Python · Biblioteca `unittest`

```python
import unittest                # Importamos la biblioteca de testing.
from ARCHIVO import FUNCIÓN    # Importamos el código a testear.

# Creamos una 'clase' donde incluiremos todos los tests.
class TestNOMBRE(unittest.TestCase):
    # Definimos los casos de test, agrupando los ejemplos de
    # alguna manera significativa.

    def test_XX(self):
        self.assertEqual(FUNCIÓN(ENTRADA1), SALIDA1)
        self.assertEqual(FUNCIÓN(ENTRADA2), SALIDA2)

    def test_YY(self):
        self.assertEqual(FUNCIÓN(ENTRADA3), SALIDA3)
        [...]

unittest.main()      # Una vez definido el test, lo ejecutamos.
```

donde `ARCHIVO` contiene el código de la `FUNCIÓN` a testear; `TestNOMBRE` es el nombre elegido para el test; `test_*` son nombres para los casos de test; y `ENTRADAi`/`SALIDAi` son los ejemplos de uso que definidos.

Además de `assertEqual`, hay otras operaciones útiles, como `assertGreater`, `assertTrue` o `assertListEqual`, que iremos viendo en la materia.

Documentación: https://docs.python.org/3/library/unittest.html

Testing de unidad · Ejemplo

```
1  import unittest                  # Importamos la biblioteca de testing
2  from ejemplo1 import cant_vocales  # Importamos el codigo a testear
3
4  # Creamos una 'clase' donde incluiremos todos los tests.
5  class TestCantVocales(unittest.TestCase):
6      # Definimos los casos de test, agrupando los ejemplos de
7      # alguna manera significativa.
8
9      def test_vacío(self):
10         self.assertEqual(cant_vocales(''), 0)
11
12     def test_vocales(self):
13         self.assertEqual(cant_vocales('aáAÁ'), 4)
14         self.assertEqual(cant_vocales('aeiouAEIOUáéíóúüÁÉÍÓÚÜ'),22)
15
16     def test_consonantes(self):
17         self.assertEqual(cant_vocales('bcdfghjklmnñpqrstvwxyz'),0)
18         self.assertEqual(cant_vocales('BCDFGHJKLMNÑPQRSTVWXYZ'),0)
19
20     def test_puntuacion(self):
21         self.assertEqual(cant_vocales(',.;:¿?¡!-\'"'), 0)
22
23     def test_repetidas(self):
24         self.assertEqual(cant_vocales('eeeee'), 5)
25
26 unittest.main()     # Una vez definido el test, lo ejecutamos.
```

Testing de unidad · Otro ejemplo

También habíamos especificado `volumen_cilindro(r:float, h:float)` $\rightarrow$ `float` y construido este conjunto de ejemplos de uso:

r	**h**	**Resultado**	Criterio/comentarios
1.0	1.0	3.14159	valores enteros
10.0	10.0	3141.5927	valores enteros
12.345	6.789	3250.4080	valores con parte decimal
0.1	0.1	0.0031416	valores con parte decimal

Nuevamente, armamos un testing de unidad en Python usando estos ejemplos.

Pero ahora, hay que tener cuidado porque nunca conviene comparar valores de tipo `float` por igualdad. Entonces, en lugar de `assertEqual`, usamos `assertAlmostEqual`.

Testing de unidad · Otro ejemplo

`assertAlmostEqual(x, y, places=N)` considera solo `N` digítos decimales.

Computa `x-y`, redondea a `N` dígitos decimales y compara el resultado a cero.

Doc: https://docs.python.org/3/library/unittest.html

```
1   import unittest                          # Importamos la biblioteca de testing.
2   from ejemplo2 import volumen_cilindro    # Importamos el codigo a testear.
3
4   # Creamos una 'clase' donde incluiremos todos los tests.
5   class TestVolumenCilindro(unittest.TestCase):
6     # Definimos los casos de test, agrupando los ejemplos de
7     # alguna manera significativa.
8
9     def test_valores_enteros(self):
10      self.assertAlmostEqual(volumen_cilindro(1.0, 1.0), 3.14159, places=3)
11      self.assertAlmostEqual(volumen_cilindro(10.0, 10.0), 3141.5927, places=3)
12
13    def test_valores_con_parte_decimal(self):
14      self.assertAlmostEqual(volumen_cilindro(12.345, 6.789), 3250.4080, places=3)
15      self.assertAlmostEqual(volumen_cilindro(0.1, 0.1), 0.0031416, places=3)
16
17  unittest.main()     # Una vez definido el test, lo ejecutamos.
```

Repaso de la clase de hoy

- Tipos de errores que pueden ocurrir en un programa:
  1. Errores de sintaxis
  2. Errores en tiempo de ejecución
  3. Errores semánticos
- La calidad del software depende de:
  1. Buenas prácticas de programación
  2. Testing
  3. Verificación entre pares
- Testing de unidad
  - Casos de test, biblioteca `unittest` en Python.

Bibliografía complementaria:

- APPP2, apéndice A.
- HTCSP3, apéndice A.

Con lo visto, ya pueden resolver toda la Guía de Ejercicios 2.