

Introducción a las ciencias de la computación *y programación en Python*

Pruebas, depuración y manejo de excepciones

Rodrigo Chang

Banco de Guatemala



Rodrigo Chang <rrcp@banguat.gob.gt>
Este material está construido a partir de modificaciones al material provisto por Ana Bell, Eric Grimson y John Guttag para el curso 6.0001 *Introduction to Computer Science and Programming in Python*, otoño 2016, Massachusetts Institute of Technology: MIT OpenCourseWare, <https://ocw.mit.edu>. Licencia: Creative Commons BY-NC-SA.

“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.”

— *John Woods*

Veremos algunos conceptos y buenas prácticas para realizar pruebas y depuración.

Veremos algunas sentencias de **excepción** y manejo de errores.

Alta calidad: una analogía

Supongamos que queremos hacer una sopa, pero algunos **insectos** caen del techo. ¿Qué harías?

- Revisar la sopa a ver si hay bichos.
 - Pruebas.
- Mantener la tapa cerrada.
 - Programación defensiva.
- Limpiar la cocina.
 - Eliminamos la fuente de los bichos.

6 STAGES OF DEBUGGING

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?

- **Programación defensiva**

- Escribir **especificaciones** para las funciones.
- **Modularizar** los programas.
- Revisar **condiciones** sobre entradas y salidas (*assertions*).

- **Validación y pruebas**

- **Comparar** pares de entradas y salidas para verificar las especificaciones.
- ¡No está funcionando!
- Pensar: ¿cómo puedo hacer que falle mi programa?

- **Depuración (*debugging*)**

- **Estudiar eventos** que han llevado a errores.
- Pensar: ¿por qué no está funcionando?
- ¿Cómo puedo arreglar mi programa?

Programación defensiva

- Desde el **principio**, **diseñar** el código para facilitar esta parte.
- Diseñar el programa en **módulos** que puedan ser probados y depurados individualmente.
- **Documentar las restricciones** sobre módulos:
 - ¿Cuál se espera que sea la entrada?
 - ¿Cuál se espera que sea la salida?
- **Documentar supuestos** detrás del código.

¿Cuándo se debe probar un código?

- Asegurarse que el código **pueda ejecutarse**.
 - Eliminar errores de sintaxis, semántica estática.
 - El intérprete de Python y los editores ayudan en esta tarea.
- Contar con un **conjunto de prueba**.
 - Conjunto de entrada del cual conocemos la salida esperada.

Clases de pruebas

- **Pruebas de unidad**

- Validar cada pieza de un programa.
- **Probar cada función** de forma separada.

- **Pruebas de regresión**

- Añadir casos de pruebas conforme se encuentran *bugs*.
- **Validar** errores reintroducidos y previamente arreglados.

- **Pruebas de integración**

- ¿Cómo unir todo el programa?, ¿funciona en conjunto?
- No se debe apresurar en este paso.

Tipos de pruebas

- **Intuición** acerca de los límites del programa.

```
def is_bigger(x, y):  
    ''' Assumes x and y are ints  
    Returns True if y is less than x, else False '''
```

- ¿alguna partición natural para probar esta función?
- Si no hay particiones naturales \Rightarrow **pruebas aleatorias**.
 - Más pruebas \Rightarrow mayor prob. de éxito.
 - Método no ideal.
- **Prueba de caja negra**
 - Explorar casos a través de la especificación.
- **Prueba de caja de cristal**
 - Explorar casos a través del conocimiento del código.

Pruebas de caja negra

```
def sqrt(x, eps):  
    ''' Assumes x, eps floats, x >= 0, eps > 0  
    Returns res such that x-eps <= res*res <= x+eps '''
```

- Diseñada **sin conocimiento** del código.
- Puede ser realizada por alguien distinto al programador.
- Puede ser **reutilizada** aún si la implementación cambia.
- Algunos caminos:
 - Casos de prueba en particiones naturales.
 - Considera **condiciones límite**: listas vacías, listas *singleton*, números muy grandes o muy pequeños, etc.

Pruebas de caja negra

```
def sqrt(x, eps):  
    """ Assumes x, eps floats, x >= 0, eps > 0  
    Returns res such that x-eps <= res*res <= x+eps """
```

CASE	x	eps
boundary	0	0.0001
perfect square	25	0.0001
less than 1	0.05	0.0001
irrational square root	2	0.0001
extremes	2	1.0/2.0**64.0
extremes	1.0/2.0**64.0	1.0/2.0**64.0
extremes	2.0**64.0	1.0/2.0**64.0
extremes	1.0/2.0**64.0	2.0**64.0
extremes	2.0**64.0	2.0**64.0

Pruebas de caja de cristal

- Utilizamos **el código** para construir los casos de prueba.
- Pueden ser **completas** si se explora cada ramificación del código al menos una vez.
- Desventajas:
 - Muchas iteraciones.
 - Nos podemos saltar alguna rama.
- Guía para diseñarlas:
 - Revisar condicionales: ¿están todos los posibles casos?
 - Revisar ciclos: ¿entra al ciclo?, ¿por qué se ejecuta una vez, o más de una vez?
 - Tipos de datos: ¿Devuelve el tipo esperado?, ¿se tuvo cuidado al operar las variables?

```
def abs(x):  
    """ Assumes x is an int  
    Returns x if x>=0 and -x otherwise """  
    if x < -1:  
        return -x  
    else:  
        return x
```

- Aún si la prueba es **completa**, podríamos saltarnos un *bug*.
- ¿Qué pasa con los casos 2 y -2?
- ¿Qué pasa con `abs(-1)`?
- Se deben probar casos **límite**.

Depuración

- Curva de aprendizaje pronunciada.
- La meta es tener un programa libre de errores.
- **Herramientas**
 - **Cerebro**: ayuda a cazar **sistemáticamente** el error.
 - Módulo **pdb**.
 - Visual Studio Code, spyder.
 - Instrucciones **print** (¡no abusar!)

Depuración con `print`

- Buena forma de probar hipótesis.
- **¿Cuándo?**
 - Entrar a funciones.
 - Parámetros
 - Resultados intermedios, finales.
- **Método de bisección**
 - `print` al centro del código.
 - Los errores dan una idea de **dónde** puede estar el error.



Pasos de depuración

- **Estudiar** el código fuente:
 - ¿Cómo obtuve el resultado inesperado?
 - ¿Es parte de alguna familia de errores?
- **Método científico:**
 - Estudiar los datos disponibles.
 - Formar una hipótesis.
 - Experimento.
 - Escoger una entrada simple para probar.

Familias de errores

- trying to access beyond the limits of a list

`test = [1,2,3]` then `test[4]` → `IndexError`

- trying to convert an inappropriate type

`int(test)` → `TypeError`

- referencing a non-existent variable

`a` → `NameError`

- mixing data types without appropriate coercion

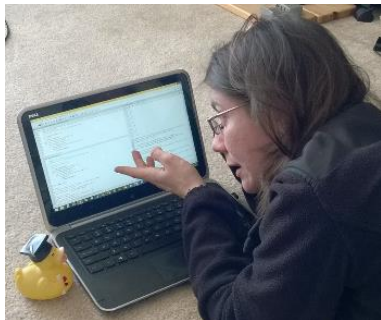
`'3'/4` → `TypeError`

- forgetting to close parenthesis, quotation, etc.

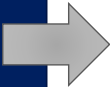
`a = len([1,2,3]`
`print(a)` → `SyntaxError`

Errores de lógica : difíciles


- **think** before writing new code
- **draw** pictures, take a break
- **explain** the code to
 - someone else
 - a rubber ducky



¿Qué no hacer?, ¿qué sí?

- Write entire program
 - Test entire program
 - Debug entire program
- 

- Write a function
- Test the function, debug the function
- Write a function
- Test the function, debug the function
- *** Do integration testing ***

- Change code
 - Remember where bug was
 - Test code
 - Forget where bug was or what change you made
 - Panic
- 

- Backup code
- Change code
- Write down potential bug in a comment
- Test code
- Compare new version with old version

Excepciones

- what happens when procedure execution hits an **unexpected condition**?

- get an **exception**... to what was expected

- trying to access beyond list limits

```
test = [1,7,4]
```

```
test[4]
```

→ `IndexError`

- trying to convert an inappropriate type

```
int(test)
```

→ `TypeError`

- referencing a non-existing variable

```
a
```

→ `NameError`

- mixing data types without coercion

```
'a' / 4
```

→ `TypeError`

- already seen common error types:
 - `SyntaxError`: Python can't parse program
 - `NameError`: local or global name not found
 - `AttributeError`: attribute reference fails
 - `TypeError`: operand doesn't have correct type
 - `ValueError`: operand type okay, but value is illegal
 - `IOError`: IO system reports malfunction (e.g. file not found)

Manejo de excepciones

- Python provee bloques de **manejo** de excepciones (exceptions).

```
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
except:
    print("Bug in user input.")
```

- Las **excepciones** que surjan (*raised by*) en el cuerpo de una sentencia **try** son manejadas por el cuerpo de **except** mientras **la ejecución continúa**.

Excepciones específicas

- Es posible especificar cláusulas de `except` para una excepción en particular.

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

Otras excepciones

- `else`:
 - Se ejecuta si la ejecución de `try` se completa sin excepciones.
- `finally`:
 - Se ejecuta **siempre** después de `try`, `else` y `except` o si ejecuta `break`, `continue` o `return`.
 - **Útil** para limpiar el código con instrucciones que de todas formas deben ejecutarse sin importar el resultado, como cerrar un archivo, conexión HTTP, etc.

¿Qué hacer con las excepciones?

- **Manejarlas silenciosamente**

- Sustituir valores por defecto, solamente continuar con el código.
- ¡Mala idea! el usuario no tiene advertencias.

- Devolver un valor para **representar** el error.

- ¿Qué valor escoger?
- Complica el resto del código.

- Detener la ejecución, **mostrar una condición de error:**

- En Python: **raise**

```
raise Exception("string descriptivo")
```

Excepciones: ejemplo

```
def get_ratios(L1, L2):  
    """ Assumes: L1, L2 lists of equal length of numbers  
    Returns: a list containing L1[i]/L2[i] """  
    ratios = []  
    for index in range(len(L1)):  
        try:  
            ratios.append(L1[index]/L2[index])  
        except ZeroDivisionError:  
            ratios.append(float('nan'))  
            #nan = not a number  
        except:  
            raise ValueError('called with bad arg')  
    return ratios
```

Excepciones: otro ejemplo

- assume we are **given a class list** for a subject: each entry is a list of two parts
 - a list of first and last name for a student
 - a list of grades on assignments

```
test_grades = [['peter', 'parker'], [80.0, 70.0, 85.0]],  
               [['bruce', 'wayne'], [100.0, 80.0, 74.0]]
```

- create a **new class list**, with name, grades, and an average

```
[[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],  
 [['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]
```

Excepciones: otro ejemplo

```
def get_stats(class_list):  
    new_stats = []  
    for elt in class_list:  
        new_stats.append([elt[0], elt[1], avg(elt[1])])  
    return new_stats  
  
def avg(grades):  
    return sum(grades)/len(grades)
```

Excepciones: otro ejemplo

- if one or more students **don't have any grades**, get an error

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],  
               [['bruce', 'wayne'], [10.0, 8.0, 74.0]],  
               [['captain', 'america'], [8.0, 10.0, 96.0]],  
               [['deadpool'], []]]
```

- **get** ZeroDivisionError: float division by zero because try to

```
return sum(grades)/len(grades)
```

length is 0

Excepciones: otro ejemplo

- decide to **notify** that something went wrong with a msg

```
def avg(grades):  
    try:  
        return sum(grades)/len(grades)  
    except ZeroDivisionError:  
        print('warning: no grades data')
```

- running on some test data gives

```
warning: no grades data
```

flagged the error

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],  
 [['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],  
 [['captain', 'america'], [8.0, 10.0, 96.0], 17.5],  
 [['deadpool'], [], None]]
```

*because avg did
not return anything
in the except*

Excepciones: otro ejemplo

- decide that a student with no grades gets a **zero**

```
def avg(grades):  
    try:  
        return sum(grades)/len(grades)  
    except ZeroDivisionError:  
        print('warning: no grades data')  
        return 0.0
```

- running on some test data gives

```
warning: no grades data
```

still flag the error

```
[[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],  
 [['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],  
 [['captain', 'america'], [8.0, 10.0, 96.0], 17.5],  
 [['deadpool'], [], 0.0]]
```

now avg returns 0

Uso de `assert`

- Cuando quieres estar **seguro** de que los supuestos del cómputo son como se esperaba.
- Utilizamos una cláusula `assert` para generar una excepción `AssertionError` si no se verifican los supuestos.
- Este es un ejemplo de buena **programación defensiva**.

Ejemplo

```
def avg(grades):  
    """ grades : grades list """  
    assert len(grades) != 0, 'no grades data'  
    return sum(grades)/len(grades)
```

- Esto lleva a una excepción de tipo `AssertionError` si se da una lista vacía de notas.
- De lo contrario, ¡OK!

Assertions como programación defensiva

- Las afirmaciones no permiten que el programador controle respuesta a condiciones inesperadas.
- Garantizan que la **ejecución se detenga** siempre que condición no se cumple.
- Normalmente se usan para **verificar las entradas** a las funciones, pero pueden ser usadas en cualquier lugar.
- Pueden usarse para **verificar las salidas** de una función para evitar propagar malos valores.
- Pueden facilitar la depuración de un error.

¿Dónde usarlas?

- El objetivo es encontrar errores tan pronto como se introduzcan.
- **Suplemento** a la etapa de pruebas.
- Levantar **excepciones** si el usuario provee datos incorrectos.
- También permiten:
 - Verificar **tipos** de los argumentos o valores.
 - Revisar **anomalías** en las estructuras de datos.
 - Revisar **restricciones** en los valores de retorno.
 - Revisar **violaciones** en las restricciones de un procedimiento.