Programación

Tema 7. Excepciones, Validación y Depuración

Javier González Villa
David Lázaro Urrutia
Departamento de Matemática Aplicada
y Ciencias de la Computación

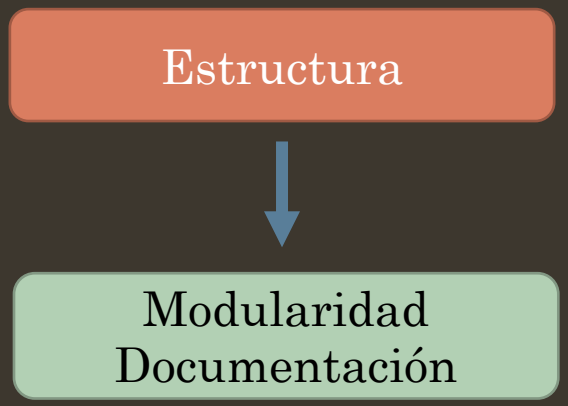
Contenidos

1. Fuentes de error
2. Validación
    1. Tipos de pruebas
    2. Enfoques de validación
3. Depuración de código
    1. Enfoques de depuración
4. Excepciones y afirmaciones

1. Fuentes de error

- Principalmente en programación existen tres fuentes fundamentales de error, las cuales pueden ser abordadas desde diferentes enfoques, pero siempre requieren de una estructura ordenada de actuación que garantice la calidad del código desarrollado.

1. Fuentes de error

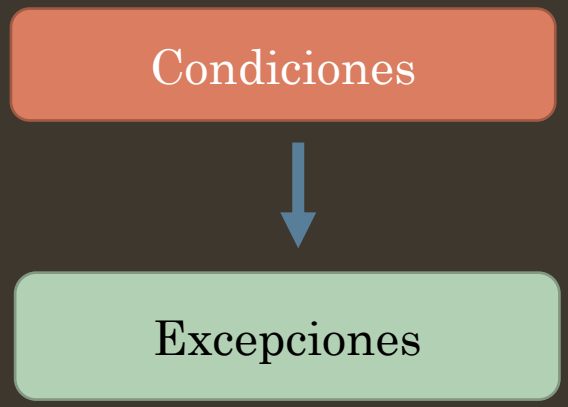


- Los fallos de estructura suceden al no plantear de manera inicial como se va a desarrollar el programa y cómo se llevará a cabo la implementación.
- Suelen producirse cuando se comienza a programar antes de pensar y estructurar el trabajo de manera que se va produciendo código sobre la marcha.
- Para resolver esto es necesario seguir un paradigma de Modularidad, es decir fragmentar nuestro código en porciones más pequeñas como funciones, las cuales son más asequibles de programar y validar.
- Por otro lado, es muy recomendable documentar en todo momento lo que se desarrolla de cara a futuros cambios.

1. Fuentes de error

- Tras implementar nuestro código de manera correcta (sin fallos de ejecución) y probar nuestro código nos damos cuenta de que no provee de los resultados esperados.
- Cuando queremos ver si nuestro código es capaz de proveer de resultados correctos para un posible abanico de entradas de datos.
- Se puede utilizar técnicas de validación para comprobar el correcto funcionamiento de nuestro programa en diferentes escenarios.
- Se puede utilizar técnicas de depuración en caso de no encontrar la fuente del error.

1. Fuentes de error



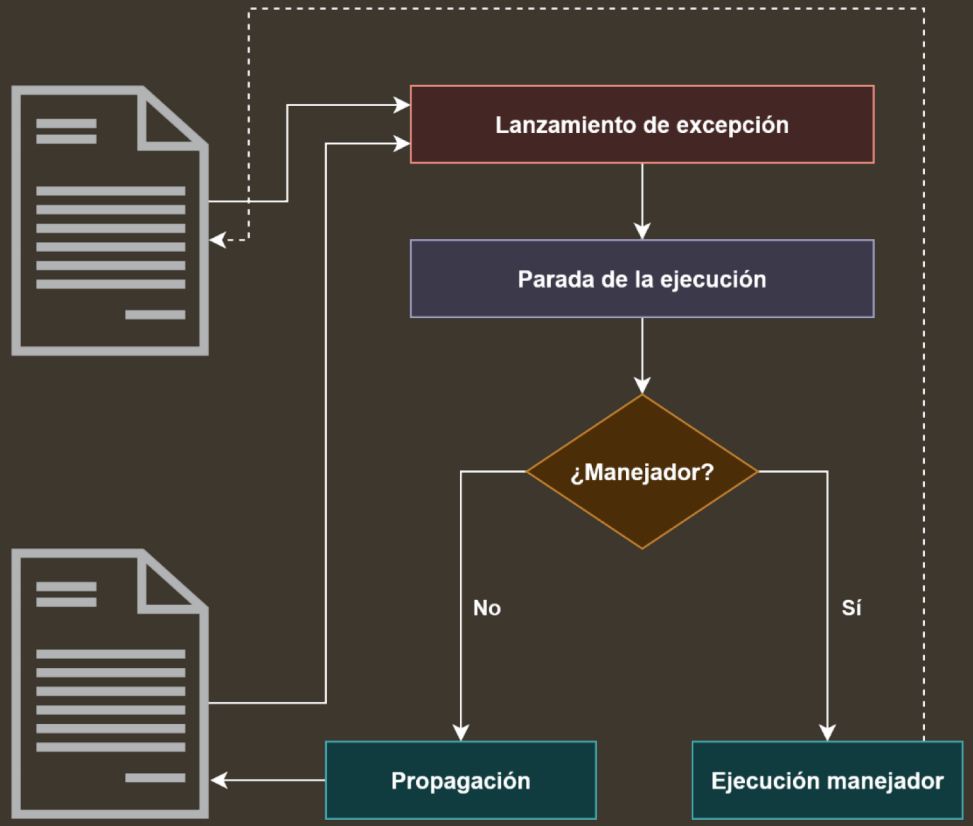
- Bajo ciertas condiciones un programa que funciona correctamente puede producir errores conocidos.
- En este caso dependiendo del tratamiento que se le quiera dar a dichos errores se pueden tener diferentes técnicas de solución.
- Una de las más comunes es el manejo de excepciones con las cuales podemos garantizar que nuestro código funcione bajo condiciones de excepción.
- También se pueden implementar técnicas de control de datos de entrada (para prevenir errores) o de retorno (para evitar propagación de errores).

1. Fuentes de error

- Algunas de los errores más comunes o excepciones que se pueden encontrar son los siguientes:
  - TypeError: operaciones o función a tipos inapropiados.
  - ZeroDivisionError: división entre cero.
  - OverflowError: números demasiado grandes.
  - IndexError: acceso a un valor de una secuencia que no existe.
  - KeyError: acceso a una clave de diccionario que no existe.
  - FileNotFoundError: intento de abrir fichero que no existe.
  - ImportError: fallo en la importación de un módulo.
  - NameError: acceso a una variable no declarada.
  - SyntaxError: fallos en la escritura del código (paréntesis, comas, corchetes, etc.)
- Aunque los errores más difíciles de solucionar son los que no son mostrados por el intérprete de Python.

1. Fuentes de error

- El flujo de excepciones o errores:



2. Validación

- Proceso necesario tras la implementación de un código para comprobar que el funcionamiento del mismo es correcto.
- Trata de validar o intentar abarcar todos los escenarios en los que se va a encontrar nuestras implementaciones para evitar posibles fallos inesperados.
- En ciertas ocasiones es necesario seguir un proceso más estricto de validación para cumplir con estándares particulares de seguridad o de fiabilidad.
- Permite reducir la carga de trabajo y los costes al automatizar y garantizar la calidad de las implementaciones.

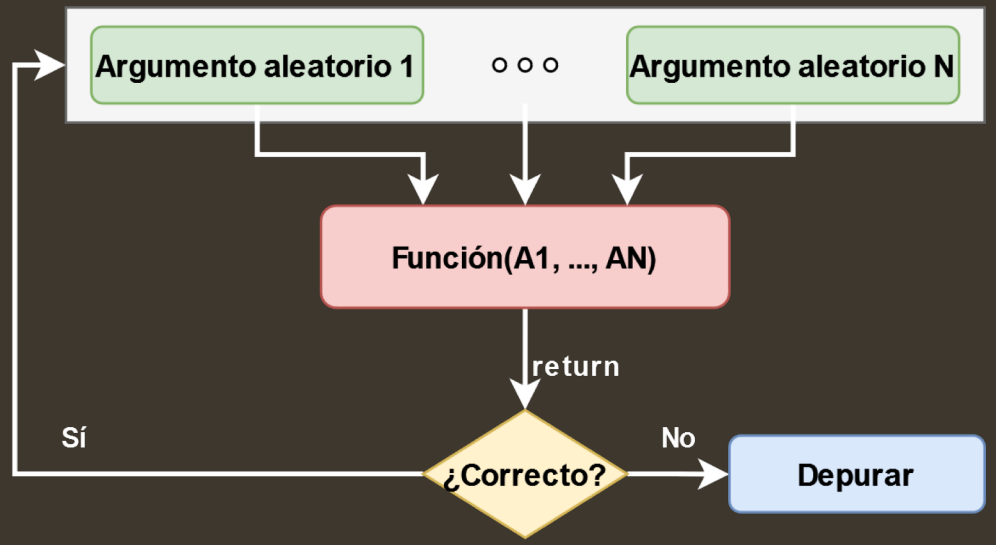
2.1. Tipos de pruebas

- Pruebas unitarias: validan cada componente o función desarrollada en nuestro código de manera individual a través de diversos enfoques.
- Pruebas de regresión: tras identificar un nuevo error, validan de nuevo de manera unitaria todo nuestro código añadiendo el nuevo error identificado.
- Pruebas funcionales: verifican las funcionalidades particulares que quiere proporcionar nuestro código independientemente del número de módulos o funciones que involucre y validando solo el resultado obtenido.
- Pruebas de integración: comprueba el correcto funcionamiento del programa de manera conjunta, estudiando las interacciones entre módulos y funciones para que todo el sistema esté en armonía.
- Pruebas de humo, de aceptación, de rendimiento, punta a punta, …

2.2. Enfoques de validación

> ***Pruebas aleatorias:*** *genera valores de prueba aleatorios con los cuales comprobar si nuestro código funciona correctamente.*

- Se utilizan si nuestros resultados no tienen unos límites claramente establecidos.
- A mayor número de pruebas mayor probabilidad de que nuestro código esté correcto.
- Se puede hacer uso de librerías de generación de número aleatorios como ***random*** en Python.



2.2. Enfoques de validación

> ***Caja negra:*** *prueba el código asumiendo no conocer su implementación y poniendo a prueba los límites de las funcionalidades esperadas.*

- No se conoce la implementación.
- Puede realizarlas alguien ajeno a la persona que creó el código (menos sesgo).
- Pueden ser reutilizadas para posteriores pruebas.
- Considera valores normales, anómalos y límite a la hora de llevar a cabo las pruebas.

Math.log()

Condición	Valor de prueba	Resultado
Negativo	-10	ValueError
Positivo	10	2.3025
Cero	0	ValueError
Máximo	1.7976931348623157e+308	709.7827
Mínimo	-1.7976931348623157e+308	ValueError
Límite	1.7976931348623157e+310	inf

2.2. Enfoques de validación

Caja blanca: *parte de la premisa de conocer código completamente e ir probando línea a línea todo el código, recorriendo cada bifurcación dentro del flujo de ejecución.*

- Se requiere conocer explícitamente el código a validar.
- Las entradas de prueba son diseñadas para recorrer cada iteración y ramificación del código.
- Es más tediosa pero más exhaustiva ya que recorre prácticamente todas las líneas de nuestro código.
- Es difícilmente reutilizable entre diferentes implementaciones y puede no contemplar casos límite.

```
[46]: def exponencial(x,n):
          return sum([(x**i)/math.factorial(i) for i in range(n)])

      try:
          e = exponencial(1, 1.0)
          print("Pruebas finalizadas correctamente.")
      except:
          print("Error encontrado para valores decimales.")

      Error encontrado para valores decimales.
```

2.2. Enfoques de validación

[48]:
```
import math
import random

def exponencial(x,n):
    return sum([(x**i)/math.factorial(i) for i in range(n)])

i = 0
error = False
while (i < 10000):
    minimo = -1.7976931348623157e+308
    maximo = 1.7976931348623157e+308
    rango = maximo - minimo
    x = (random.random()*rango)-minimo
    try:
        e = exponencial(x,100)
    except:
        error = True
        print("Error encontrado para el valor ",x)
        break
    finally:
        i = i + 1
if not error:
    print("Pruebas finalizadas correctamente.")
```

```
Pruebas finalizadas correctamente.
```

[46]:
```
def exponencial(x,n):
    return sum([(x**i)/math.factorial(i) for i in range(n)])

try:
    e = exponencial(1, 1.0)
    print("Pruebas finalizadas correctamente.")
except:
    print("Error encontrado para valores decimales.")
```

```
Error encontrado para valores decimales.
```

3. Depuración de código

- Si mediante el proceso de validación hemos encontrado un error que no sabemos resolver de manera inmediata, es momento de depurar el código.
- Este proceso trata de recorrer el código de manera metódica y ordenada hasta encontrar el error.
- Si hemos seguido un paradigma de modularidad, será más fácil encontrar el error ya que estará acotado dentro de una función o módulo.
- Dependiendo del error obtenido, puede requerir retomar una visión global de los datos de entrada y salida esperados así como el flujo de ejecución del código.

3.1. Enfoques de depuración

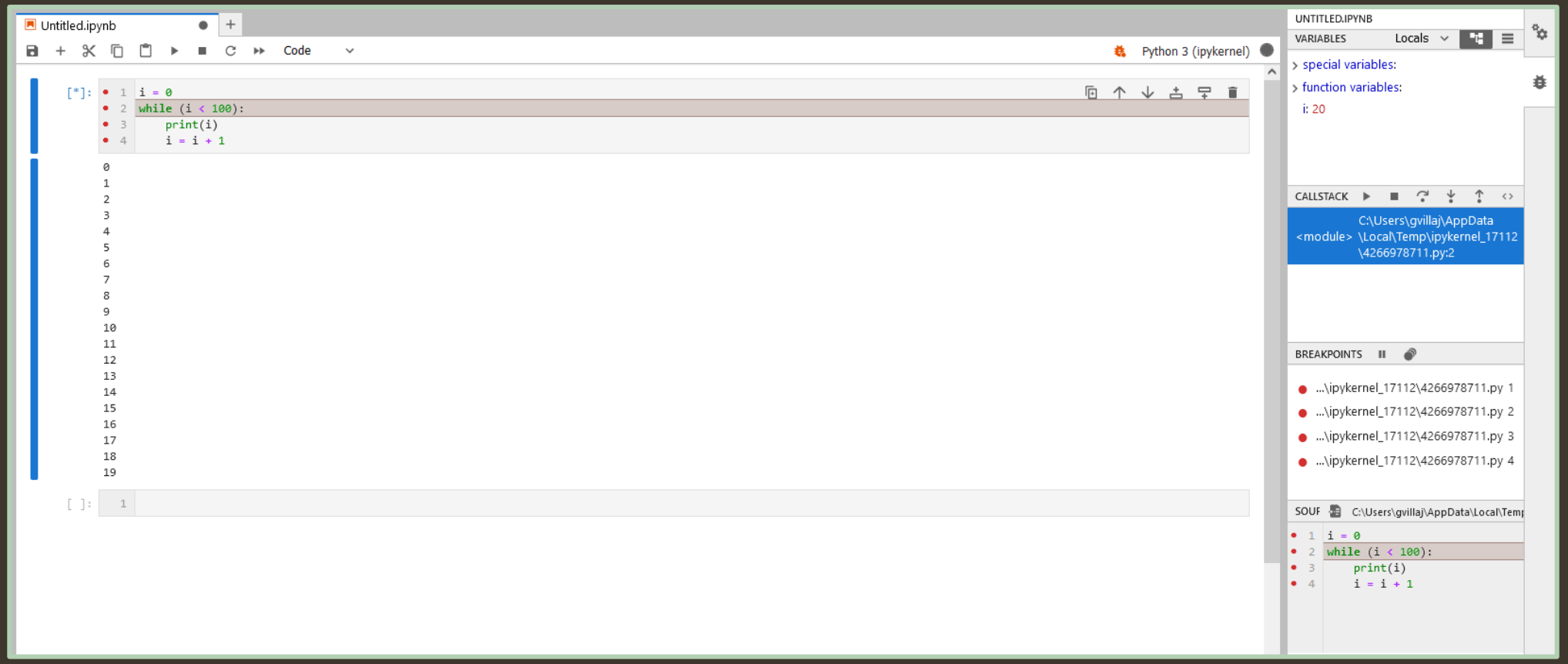
- Estudio del flujo del código: entender cómo se comporta la ejecución de nuestro código a través de diagramas. Es necesario para acotar dónde se produce el error.

- Utilizar la sentencia ***print***: Una vez se tiene una idea de donde puede estar localizado el error, utilizar sentencias que pinten el estado de las variables, puede ser una buena idea para conocer de donde proviene el error.

- Utilizar el Debugger: En caso de estar completamente perdidos y requerir de una ejecución global, se puede hacer uso del Debugger de JupyterLab para ir paso a paso ejecutando nuestro código y encontrar el error.

3.1. Enfoques de depuración

- Utilizar el Debugger y los puntos de interrupción



4. Excepciones y afirmaciones

- Se producen cuando nuestro código correctamente implementado se encuentra con una condición no esperada en su ejecución.
- Se tendrían los errores antes comentados u otros similares.
- Se pueden tener dos enfoques a la hora de prevenir estas condiciones no deseadas:

1. Programación defensiva (afirmaciones): comprobar cada parámetro y retorno de cada función o método para que no se desvíe de lo esperado (tipo, límites, número de datos, …).

2. Manejo de excepciones (excepciones): sin necesidad de comprobar los datos, una vez que se produce el error nuestro código está preparado para identificarlo y tratarlo.

4. Excepciones y afirmaciones

1. Programación defensiva (afirmaciones)

```
assert condición, “mensaje de error”
```

2. Manejo de excepciones (excepciones):

```
try:
        bloque de código donde se produce un error
except ErrorParticular:
        actuación frente al ErrorParticular
except:
        actuación frente a cualquier otro error
finally:
        bloque que siempre se ejecutará
```

4. Excepciones y afirmaciones

1. Programación defensiva (afirmaciones)

assert condición, “mensaje de error”

2. Manejo de excepciones (excepciones):

- Sustituir el valor erróneo por otro valido (no recomendable).
- Retornar un mensaje de error por pantalla.
- Parar la ejecución y relanzar la excepción para que la trate otra función.

raise Excepcion(“mensaje de error”)

4. Excepciones y afirmaciones

```
[42]: import math

def exponencial(x,n):
    try:
        return sum([(x**i)/math.factorial(i) for i in range(n)])
    except ValueError:
        print("Error en los valores introducidos.")
        return sum([(x**i)/math.factorial(i) for i in range(100)])
    except TypeError:
        print("Error en los tipos introducidos.")
        return sum([(x**i)/math.factorial(i) for i in range(100)])
    except:
        print("Vuelvo a lanzar la excepción.")
        raise ValueException("Error desconocido.")
    finally:
        print("Siempre se ejecuta!")

print(exponencial(1,100))
print(exponencial(1,'hola'))
```

```
Siempre se ejecuta!
2.7182818284590455
Error en los tipos introducidos.
Siempre se ejecuta!
2.7182818284590455
```

```
[51]: import math

def exponencial(x,n):
    assert (type(x) == float) or (type(x) == int), "Tipo de x incorrecto."
    assert (type(n) == int), "Tipo de n incorrecto."
    assert n > 0, "El valor de n debe ser positivo."

    e = sum([(x**i)/math.factorial(i) for i in range(n)])

    assert (type(e) == float) or (type(e) == int), "Tipo de resultado incorrecto."
    assert e > 0, "Valor del resultado incorrecto."

    return e

print(exponencial(1,100))
print(exponencial('hola',100))
```

```
2.7182818284590455
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9928\4241624326.py in <module>
     14
     15 print(exponencial(1,100))
---> 16 print(exponencial('hola',100))
     17 print(exponencial(1,100))

~\AppData\Local\Temp\ipykernel_9928\4241624326.py in exponencial(x, n)
      2
      3 def exponencial(x,n):
----> 4     assert (type(x) == float) or (type(x) == int), "Tipo de x incorrecto."
      5     assert (type(n) == int), "Tipo de n incorrecto."
      6     assert n > 0, "El valor de n debe ser positivo."

AssertionError: Tipo de x incorrecto.
```