Depuración de Código

Errores y Excepciones

Ya vimos como manejar excepciones utilizando la sentencia try except para evitar que nuestro programa se cierre inesperadamente. Estas excepciones las levanta Python por si solo pero también podemos levantarlas nosotros mismos.

Las excepciones ocurren con una sentencia gatillo o raise statement. En código, esta sentencia consiste de:

- La palabra reservada raise
- La llamada a la función Exception()
- Una cadena con un mensaje de error que describa la situación lo mejor posible.

Por ejemplo:

```
>>> raise Exception('Este es un mensaje de error.')
```

Si no hay ningún otro try except cubriendo esta sentencia el programa simplemente crashea mostrando el mensaje de error.
Normalmente verán un raise statement adentro de una función y el try except cubriendo el código que llamo a esa función. Ejemplo:

```
def printCaja(simbolo, ancho, altura):
  if len(simbolo) != 1:
    raise Exception('El simbolo solo puede ser 1 caracter.')
  if ancho <= 2:
    raise Exception('El ancho debe ser más grande que 2')
  if altura <= 2:
    raise Exception('La altura debe ser más grande que 2.')
  print(simbolo * ancho)
  for i in range(altura - 2):
    print(simbolo + (' ' * (ancho - 2)) + simbolo)
  print(simbolo * ancho)

for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
  try:
    printCaja(sym, w, h)
  except Exception as err:
    print('Una excepcion ocurrió: ' + str(err))
```

Consiguiendo el Traceback como cadena

Cuando Python encuentra un error, produce información a la que se le llama traceback o trazado inverso. Este traceback incluye el mensaje de error, el número de la línea que lo causó, y la secuencia que llamó la función y produjo el error. A esta secuencia se le llama la call stack, o pila de ejecución. Ejemplo:

```
def spam():
  bacon()
def bacon():
  raise Exception('Este es el mensaje de error.')
spam()
```

Python muestra por pantalla el traceback cuando se levanta una excepción y no tiene como manejarla. Pero también podemos conseguirlo en una cadena utilizando la función traceback.format_exc(). Esta función es útil cuando queremos información sobre una excepción, pero al mismo tiempo queremos que Python sepa cómo manejarla. Para poder utilizarla primero debemos importar el módulo traceback.

```
>>> import traceback
>>> try:
  raise Exception('Este es un mensaje de error.')
except:
  errorFile = open('errorInfo.txt', 'w')
  errorFile.write(traceback.format_exc())
  errorFile.close()
  print('La info del traceback fue guardada en errorInfo.txt.')
```

Assertions o Aserción

Una assertion es una manera de asegurarse que nuestro código no está haciendo algo absolutamente mal. Este control se hace con una sentencia assert y si falla entonces se levanta una excepción del tipo AssertionError. Consiste de:

- La palabra clave assert
- Una condición
- Una coma
- Una cadena a mostrar si la condición es falsa

Ejemplo:

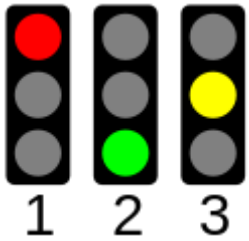
```
>>> edades = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]
>>> edades.sort()
>>> edades
>>> assert edades[0] <= edades[-1]
>>> edades = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]
>>> edades.reverse()
>>> edades
>>> assert edades[0] <= edades[-1], 'La primera edad es más grande que la última'
```

A diferencia de las excepciones, nuestro código no debe manejar sentencias assert con un try except. Si un assert falla, nuestro programa debería crashear. Al fallar tan rápido acorta el tiempo entre que creamos el bug/error y la primera vez que lo notemos. Esto a su vez disminuye la cantidad de código que tenemos que revisar para encontrar la causa de un bug/error.

En ese sentido las aserciones/assertions son errores causados por el programador, no por el usuario. Las aserciones solo deben fallar mientras estamos desarrollando un programa y un usuario jamás debería ver un assertion error en un programa terminado. Para errores que nuestro programa puede encontrar como parte de su ejecución normal (como cuando se ingresa información inválida), hay que levantar excepciones en su lugar. Si ejecutamos un Python script con el argumento –O, Python ignorará las sentencias assert. Esto nos permite disminuir el tiempo de ejecución mientras estamos desarrollando un programa para ver como se comportará una vez que este terminado.

Usando Assertions en un simulador de luces de semáforo

Digamos que estamos programando un simulador de luces de semáforo de 3 etapas con el siguiente código:




```
calleRivadavia = {'eo': 'rojo'}

calleJujuy = {'ns': 'verde'}
```

Estas 2 variables son para la intersección de Rivadavia y Jujuy. Para empezar necesitamos una función cambiarLuces(), que tomará una intersección y cambiará las luces. Y para probar las aserciones que acabamos de ver pondremos en el código algo que no queremos que ocurra nunca.

Para este ejercicio pueden utilizar la función time.sleep(t) donde t es el número de segundos que queremos detener el programa. Tendrán que importar el módulo time para poder usarla.

Logging o registro

Cuando usamos un print() en nuestro código para verificar problemas de nuestro programa estamos creando una especie de registro de lo que está haciendo nuestro programa y para resolver algún tipo de problema. El módulo logging de Python hace que esto sea más fácil, y avisarnos además de secciones de código que no se hayan ejecutado jamás.

Usando el módulo logging

```
import logging
```

```
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -  %(levelname)s
-  %(message)s')
```

Cuando Python registra un evento, crea un objeto LogRecord que contiene información acerca de ese evento. La función basicConfig() de este modulo nos permite especificar qué detalles del objeto LogRecord queremos ver y como mostrarlo.

El siguiente programa lleva un registro mientras trata de calcular el factorial de un número.

```
import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s -  %(levelname)s-
%(message)s')
logging.debug('Inicio del Programa')
def factorial(n):
   logging.debug('Comienzo de factorial(%s)'  % (n))
   total = 1
   for i in range(n + 1):
      total *= i
      logging.debug('i es ' + str(i) + ', el total es ' + str(total))
   logging.debug('Fin de factorial(%s)'  % (n))
   return total
print(factorial(5))
logging.debug('Fin del Programa')
```

```
2020-11-01 20:15:37,302 - DEBUG- Inicio del Programa
2020-11-01 20:15:37,302 - DEBUG- Comienzo de factorial(5)
2020-11-01 20:15:37,302 - DEBUG- i es 0, el total es 0
2020-11-01 20:15:37,302 - DEBUG- i es 1, el total es 0
2020-11-01 20:15:37,303 - DEBUG- i es 2, el total es 0
2020-11-01 20:15:37,303 - DEBUG- i es 3, el total es 0
2020-11-01 20:15:37,303 - DEBUG- i es 4, el total es 0
2020-11-01 20:15:37,303 - DEBUG- i es 5, el total es 0
2020-11-01 20:15:37,303 - DEBUG- Fin de factorial(5)
0
2020-11-01 20:15:37,303 - DEBUG- Fin del Programa
```

La función factorial() está devolviendo 0 como factorial de 5, lo cual está mal. El ciclo for debería estar multiplicando el valor en total por los números desde 1 a 5. Pero los mensajes del log muestran que la variable está empezando en 0 en vez de 1. Como cero por cualquier número da 0 el resto de las iteraciones tendrán problemas también. El log entonces nos dejó un rastro de migajas para saber cuál es problema con el programa.

Cambiamos la línea de for i in range(n + 1) por for i in range(1, n+1) para arreglar el programa. El log ahora se verá así:

```
2020-11-01 20:23:03,604 - DEBUG- Inicio del Programa
2020-11-01 20:23:03,604 - DEBUG- Comienzo de factorial(5)
2020-11-01 20:23:03,605 - DEBUG- i es 1, el total es 1
2020-11-01 20:23:03,605 - DEBUG- i es 2, el total es 2
```

```
2020-11-01 20:23:03,605 - DEBUG- i es 3, el total es 6
2020-11-01 20:23:03,605 - DEBUG- i es 4, el total es 24
2020-11-01 20:23:03,605 - DEBUG- i es 5, el total es 120
2020-11-01 20:23:03,605 - DEBUG- Fin de factorial(5)
120
2020-11-01 20:23:03,605 - DEBUG- Fin del Programa
```

Dejando de depurar con print() y empezando a usar logging

A partir de ahora simplemente agreguen las siguientes líneas al comienzo de sus programas:

```
import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s -
%(message)s')
```

Y ya podrán empezar a registrar eventos sin tener que utilizar print(). Lo bueno de esto es que una vez terminado el programa no hace falta eliminar todos los logging.debug() como con print(), simplemente agregamos logging.disable(logging.CRITICAL) al comienzo del programa y eso evitará que se muestre el log.

Para mensajes como “La información ingresada no es válida” o “Archivo no encontrado” está bien utilizar la función print() porque son errores dirigidos al usuario. Pero al usuario no le interesa el valor que tiene un diccionario en la 3ra iteración de nuestro ciclo for, así que no hace falta mostrárselos.

Niveles de Registro o Logging Levels

Los niveles de registro o Logging levels no permiten categorizar nuestros mensajes de log o regsitro por nivel de importancia. Hay 5 niveles que se describen en la siguiente tabla y podemos crear logs de cada nivel usando la función correspondiente al mismo.

Nivel	Función	Descripción
DEBUG	logging.debug()	El nivel más bajo, utilizado para pequeños detalles. Más que nada son útiles para diagnosticas problemas.
INFO	logging.info()	Utilizado para eventos en general o para confirmar que algo en particular está funcionando correctamente.
WARNING	logging.warning()	Utilizado para indicar que puede que exista un problema pero que no impide el funcionamiento del programa aún.
ERROR	logging.error()	Utilizado para registrar un problema que hizo que algo en el programa no funcionara bien.
CRITICAL	logging.critical()	El nivel más alto. Utilizado para indicar un error fatal que causo o está por causar el cierre inmediato del programa.

Cada una de estas funciones toma una cadena como argumentos y nos toca a nosotros decidir que nivel utilizar.

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s -  %(message)s')
>>> logging.debug('Detalles para depurar.')
>>> logging.info('El módulo de depuración está funcionando.')
>>> logging.warning('Un mensaje de error está por ser registrado.')
>>> logging.error('Ha ocurrido un error!')
>>> logging.critical('El programa no puede recuperarse!')
```

El beneficio de utilizar niveles para nuestros logs es que más adelante podemos ocultar los niveles que menos nos importen como DEBUG o INFO al cambiar al configuración de basicConfig para mostrar niveles por arriba WARNING o ERROR solamente.

Deshabilitando el Logging

Luego de depurar un programa, ya no es necesario ver nuestros mensajes de log. Para poder ocultarlos está la función logging.disable() que toma como argumento hasta qué nivel queremos ocultar, inclusive. logging.disable(logging.CRITICAL) entonces ocultará todos los mensajes de log si la llamamos al principio del programa.

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -
%(levelname)s -  %(message)s')
>>> logging.critical(' Error crítico! Error crítico! ')
>>> logging.disable(logging.CRITICAL)
>>> logging.critical(' Error crítico! Error crítico!')
>>> logging.error('Error! Error!')
```

Guardando el Log en un archivo

En lugar de mostrar nuestro log por pantalla podémos guardarlo en un archive. Uno de los argumentos de logging.basicConfig() nos permite hacer esto.

```
import logging
logging.basicConfig(filename='mensajesLog.txt', level=logging.DEBUG, format='
%(asctime)s -  %(levelname)s -  %(message)s')
```

Los mensajes se guardarán en mensajesLog.txt. Esto evita que se muestren demasiados mensajes por pantalla que nos impidan ver el funcionamiento normal del programa.