



El futuro digital
es de todos

MinTIC

Ciclo 1

Fundamentos de programación con Python

Sesión 19: Manejo de Excepciones en Python

Programa Ciencias de la Computación e Inteligencia Artificial
Escuela de Ciencias Exactas e Ingeniería
Universidad Sergio Arboleda
Bogotá



UNIVERSIDAD
SERGIO ARBOLEDA

Mision
TIC 2022



Agenda

1. Errores de sintaxis
2. Excepciones
3. Manejo de excepciones
4. Planteamiento de excepciones
5. Excepciones definidas por el usuario
6. Definición de las acciones de limpieza
7. Ejercicios propuestos





1. Errores de sintaxis

- Los errores de sintaxis, también conocidos como errores de análisis sintáctico, son quizás el tipo de error más común que se encuentra mientras se aprende Python.
- El analizador sintáctico repite la línea infractora y muestra una pequeña "flecha" que señala el punto más temprano de la línea donde se detectó el error. El error se detecta en el token que precede a la flecha. El nombre del archivo y el número de línea se imprimen para que sepas dónde buscar en caso de que la entrada provenga de un script.

```
>>> while True print 'Hello world'
File "<stdin>", line 1, in ?
    while True print 'Hello world'
                        ^
SyntaxError: invalid syntax
```





2. Excepciones

- Incluso si una sentencia o expresión es sintácticamente correcta, puede causar un error cuando se intenta ejecutarla.
- Los errores detectados durante la ejecución se denominan excepciones y no son incondicionalmente fatales. Sin embargo, la mayoría de las excepciones no son manejadas por los programas y dan lugar a mensajes de error como "no se puede dividir por cero" o "no se pueden concatenar los objetos 'str' e 'int'".

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```





3. Manejo de excepciones

- Es posible escribir programas que manejen excepciones seleccionadas. Considere lo siguiente, donde una interrupción generada por el usuario es señalada lanzando la excepción **KeyboardInterrupt**
- Primero se ejecuta la sentencia **'try'** hasta que se produce una excepción, en cuyo caso se salta el resto de la sentencia **'try'** y se ejecuta la sentencia **'except'** (dependiendo del tipo de excepción), y la ejecución continúa. Si se produce una excepción que no coincide con la excepción nombrada en la sentencia **'except'**, se pasa a las sentencias **'try'** externas; si no se encuentra un manejador, es una excepción no manejada y la ejecución se detiene.

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number. Try again..."
```





3. Manejo de excepciones

- La última sentencia **except** (cuando se declaran muchas) puede omitir el nombre de la(s) excepción(es), para servir de comodín. Esto hace que sea muy fácil enmascarar un verdadero error de programación. También puede utilizarse para imprimir un mensaje de error y luego volver a lanzar la excepción.
- La sentencia **try-except** tiene una cláusula **else** opcional que, cuando está presente, debe seguir a todas las cláusulas **except**. Es útil para el código que debe ejecutarse si la cláusula **try** no lanza una excepción.

```
for arg in sys.argv[1:]:  
    try:  
        f = open(arg, 'r')  
    except IOError:  
        print 'cannot open', arg  
    else:  
        print arg, 'has', len(f.readlines()), 'lines'  
        f.close()
```





4. Planteamiento de excepciones

- La sentencia **raise** permite al programador forzar la ocurrencia de una excepción específica.

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

- El único argumento de la sentencia **raise** indica la excepción que se va a lanzar.
- Una forma más sencilla de la sentencia **raise** permite relanzar la excepción (si no se quiere manejar):

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```





5. Excepciones definidas por el usuario

- Los programas pueden nombrar sus propias excepciones creando una nueva clase de excepción. Estas derivan de la clase Exception, ya sea directa o indirectamente.
- Aquí, el `def__init__()` de Exception ha sido sobrescrito. El nuevo comportamiento simplemente crea el atributo value.

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    __main__.MyError: 'oops!'
```





6. Definición de las acciones de limpieza

- La sentencia **try** tiene otra cláusula opcional cuyo objetivo es definir las acciones de limpieza que deben ejecutarse en cualquier circunstancia.
- Una cláusula **finally** se ejecuta antes de salir de la sentencia **try**, tanto si se ha producido una excepción como si no. Cuando se ha producido una excepción en la cláusula **try** y no ha sido manejada por una cláusula **except**, se vuelve a lanzar después de que se haya ejecutado la cláusula **finally**. La cláusula **finally** también se ejecuta "a la salida" cuando se sale de cualquier otra cláusula de la sentencia **try** utilizando **break/continue/return**.

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
```





➤ Acciones de limpieza predefinidas

- Algunos objetos definen acciones de limpieza estándar que se llevan a cabo cuando el objeto ya no es necesario, independientemente de si la operación que utiliza el objeto ha tenido éxito o no.

```
for line in open("myfile.txt"):  
    print line
```

- El problema de este código es que deja el archivo abierto durante un tiempo indefinido después de que el código haya terminado de ejecutarse.
- La sentencia **'with'** permite utilizar objetos como los archivos de forma que se garantice que siempre se limpien de forma rápida y correcta.

```
with open("myfile.txt") as f:  
    for line in f:  
        print line
```





7. Ejercicios propuestos

- Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

```
resultado = 10/0
```

- Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

```
lista = [1, 2, 3, 4, 5]  
lista[10]
```





7. Ejercicios propuestos

- Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

```
colores = { 'rojo': 'red', 'verde': 'green', 'negro': 'black' }  
colores['blanco']
```

- Localiza el error en el siguiente bloque de código. Crea una excepción para evitar que el programa se bloquee y además explica en un mensaje al usuario la causa y/o solución:

```
resultado = 15 + "20"
```





Preguntas

