PyLearn_PyTutorial_8_ErrorsExceptions

May 2, 2020

#

Learning Python

1 The Python Tutorial -> Errors and Exceptions

Link: https://docs.python.org/3/tutorial/errors.html

1.1 Handling Exceptions

- Built-in Exceptions https://docs.python.org/3/library/exceptions.html#bltin-exceptions

[7]:
```
while True:
    try:
        x = int(input("Please enter a number: "))
        print("You entered " + str(x))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

```
Please enter a number:  t

Oops!  That was no valid number.  Try again…

Please enter a number:  5

You entered 5
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an `except` clause listing a derived class is not compatible with a base class).

[9]:
```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

```
B
C
D
```

- When `B` is raised, the derived class `D` and `C` of `B` in the `except` clause are not compatible with the base class `B`.
- Similarly, When `C` is raised, the derived class `D` of `C` in the `except` clause is not compatible with the base class `C`.

[10]:
```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except B:
        print("B")
    except C:
        print("C")
    except D:
        print("D")
```

```
B
B
B
```

When `C` or `D` are raised, the base class `B` in the `except` clause is compatible with the derived class `C` or `D`.

The last except clause may omit the exception name(s), to serve as a wildcard

[18]:
```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Hey, you got an unexpected error:", sys.exc_info()[0])
    raise    # re-raise it
```

```
OS error: [Errno 2] No such file or directory: 'myfile.txt'
```

[19]:
```
import sys

try:
    Z = some_unknown_function()
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Hey, you got an unexpected error:", sys.exc_info()[0])
    raise    # re-raise it
```

```
Hey, you got an unexpected error: <class 'NameError'>


      ␣
↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-19-c627f0b49709> in <module>
          2
          3 try:
    ----> 4     Z = some_unknown_function()
          5     f = open('myfile.txt')
          6     s = f.readline()
        NameError: name 'some_unknown_function' is not defined
```

```
[51]: import sys

      def fun(x, y):
          try:
              z = x / y
              print(x, "divide by", y, "is", z)
          except ZeroDivisionError:
              print(x, "cannot be divided by", y)
          except:
              print("Hey, you got an unexpected error:")
              print("   ", sys.exc_info()[0])
              print("   ", sys.exc_info()[1])
              print("   ", sys.exc_info()[2])
              raise    # re-raise it
      fun(4, 3)
      fun(1, 0)
      fun('4', '3')
```

```
4 divide by 3 is 1.3333333333333333
1 cannot be divided by 0
Hey, you got an unexpected error:
    <class 'TypeError'>
    unsupported operand type(s) for /: 'str' and 'str'
    <traceback object at 0x7fb284212ac0>


      ␣
 ↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-51-845d04dbc7e2> in <module>
         15 fun(4, 3)
         16 fun(1, 0)
    ---> 17 fun('4', '3')


        <ipython-input-51-845d04dbc7e2> in fun(x, y)
          3 def fun(x, y):
          4     try:
    ----> 5         z = x / y
          6         print(x, "divide by", y, "is", z)
          7     except ZeroDivisionError:
```

```
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Optional `else` clause, which, when present, must follow all `except` clauses. It is useful for code that must be executed if the `try` clause does not raise an exception.
The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try` … `except` statement.

[71]:
```
def fun(x, y):
    try:
        z = x / y
    except ZeroDivisionError:
        print(x, "cannot be divided by", y)
    else:
        print(x, "divided by", y, "is", z)
        return z

x = 1
y = 0
z = fun(x, y)
print("fun(", x, ",", y, ") returns ", z)

x = 4
y = 3
z = fun(x, y)
print("fun(", x, ",", y, ") returns ", z)
```

```
1 cannot be divided by 0
fun( 1 , 0 ) returns  None
4 divided by 3 is 1.3333333333333333
fun( 4 , 3 ) returns  1.3333333333333333
```

Exception handlers don't just handle exceptions if they occur immediately in the `try` clause, but also if they occur inside functions that are called (even indirectly) in the `try` clause. For example:

[75]:
```
def fun(x, y):
    z = x / y
    return z

try:
    x = 1
    y = 0
    z = fun(x, y)
```

```
except ZeroDivisionError:
    print(x, "cannot be divided by", y)
else:
    print(x, "divided by", y, "is", z)
print("fun(", x, ",", y, ") returns ", z)

try:
    x = 4
    y = 3
    z = fun(x, y)
except ZeroDivisionError:
    print(x, "cannot be divided by", y)
else:
    print(x, "divided by", y, "is", z)
print("fun(", x, ",", y, ") returns ", z)
```

```
1 cannot be divided by 0
fun( 1 , 0 ) returns  1.3333333333333333
4 divided by 3 is 1.3333333333333333
fun( 4 , 3 ) returns  1.3333333333333333
```

exception instance The `except` clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance`.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`.
One may also instantiate an exception first before raising it and add any attributes to it as desired.

[53]:
```
try:
    raise Exception('spam', 'eggs')
except Exception as inst:
    print(type(inst))    # the exception instance
    print(inst.args)     # arguments stored in .args
    print(inst)          # __str__ allows args to be printed directly,
                         # but may be overridden in exception subclasses
    x, y = inst.args     # unpack args
    print('x =', x)
    print('y =', y)
    print("Hey, you got an unexpected error:")
    print("   ", sys.exc_info()[0])
    print("   ", sys.exc_info()[1])
    print("   ", sys.exc_info()[2])
```

```
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

```
Hey, you got an unexpected error:
    <class 'Exception'>
    ('spam', 'eggs')
    <traceback object at 0x7fb28420e640>
```

1.2 Raising Exceptions

The sole argument to `raise` indicates the exception to be raised.
This must be either an **exception instance** or an **exception class** (a class that derives from `Exception`).

[76]:
```
raise NameError('HiThere')
```

```
       ␣
 ↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-76-72c183edb298> in <module>
    ----> 1 raise NameError('HiThere')


        NameError: HiThere
```

If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

[79]:
```
raise ValueError  # shorthand for 'raise ValueError()'
```

```
       ␣
 ↪---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call␣
 ↪last)

        <ipython-input-79-496f17a27c64> in <module>
    ----> 1 raise ValueError  # shorthand for 'raise ValueError()'


        ValueError:
```

shorthand for `raise ValueError()`

If we need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

[80]:
```
try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')
    raise
```

```
An exception flew by!


       ␣
↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-80-bf6ef4926f8c> in <module>
          1 try:
    ----> 2     raise NameError('HiThere')
          3 except NameError:
          4     print('An exception flew by!')
          5     raise


        NameError: HiThere
```

1.3 User-defined Exceptions

When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

[82]:
```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

1.4 Defining Clean-up Actions

Optional **finally** clause which is intended to define clean-up actions that must be executed under all circumstances.
In real world applications, the **finally** clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

[87]:
```
try:
    a = 3
finally:
    print('Goodbye, world!')
```

```
Goodbye, world!
```

[88]:
```
try:
    raise KeyboardInterrupt
finally:
    print('Goodbye, world!')
```

```
Goodbye, world!


        ␣
 ↪---------------------------------------------------------------------------

        KeyboardInterrupt                         Traceback (most recent call␣
 ↪last)

        <ipython-input-88-ca8991ac7661> in <module>
```

```
      1 try:
----> 2     raise KeyboardInterrupt
      3 finally:
      4     print('Goodbye, world!')


    KeyboardInterrupt:
```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes.
The `finally` clause runs whether or not the `try` statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `try` statement reaches a `break`, `continue` or `return` statement, the `finally` clause will execute just prior to the `break`, `continue` or `return` statement's execution.
- **If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement.**

[91]:
```python
def bool_return():
    try:
        return True
    finally:
        return False

bool_return()
```

[91]:
```
False
```

[92]:
```python
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")

divide(2, 1)
divide(2, 0)
divide("2", "1")
```

```
result is 2.0
executing finally clause
division by zero!
executing finally clause
executing finally clause
```

```
       ␣
 ↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-92-e9ae94121bc8> in <module>
         11 divide(2, 1)
         12 divide(2, 0)
    ---> 13 divide("2", "1")


        <ipython-input-92-e9ae94121bc8> in divide(x, y)
          1 def divide(x, y):
          2     try:
    ----> 3         result = x / y
          4     except ZeroDivisionError:
          5         print("division by zero!")


        TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

1.5 Predefined Clean-up Actions¶

Example:

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

After the statement is executed, the file f is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

2 END OF The Python Tutorial -> Errors and Exceptions