Exceptions

Python 2.x 2.3–2.7 , Python 3.x 3.0–3.6

Examples

Catching Exceptions

Use `try...except:` to catch exceptions. You should specify as precise an exception as you can:

```
try:
    x = 5 / 0
except ZeroDivisionError as e:
    # `e` is the exception object
    print("Got a divide by zero! The exception was:", e)
    # handle exceptional case
    x = 0
finally:
    print "The END"
    # it runs no matter what execute.
```

The exception class that is specified - in this case, `ZeroDivisionError` - catches any exception that is of that class or of any subclass of that exception.

For example, `ZeroDivisionError` is a subclass of `ArithmeticError` :

```
>>> ZeroDivisionError.__bases__
(<class 'ArithmeticError'>,)
```

And so, the following will still catch the `ZeroDivisionError` :

```
try:
    5 / 0
except ArithmeticError:
    print("Got arithmetic error")
```

Do not catch everything!

While it's often tempting to catch every `Exception` :

```
try:
    very_difficult_function()
except Exception:
    # log / try to reconnect / exit gratiously
finally:
    print "The END"
    # it runs no matter what execute.
```

Or even everything (that includes `BaseException` and all its children including `Exception`):

```
try:
    even_more_difficult_function()
except:
    pass  # do whatever needed
```

In most cases it's bad practice. It might catch more than intended, such as `SystemExit` , `KeyboardInterrupt` and `MemoryError` - each of which should generally be handled differently than usual system or logic errors. It also means there's no clear understanding for what the internal code may do wrong and how to recover properly from that condition. If you're catching every error, you wont know what error occurred or how to fix it.

Usually these constructs are used at the very outer level of the program, and will log the details of the error so that the bug can be fixed, or the error can be handled more specifically.

Catching multiple exceptions

There are a few ways to catch multiple exceptions .

The first is by creating a tuple of the exception types you wish to catch and handle in the same manner. This example will cause the code to ignore `KeyError` and `AttributeError` exceptions.

```
try:
    d = {}
```

```
    a = d[1]
    b = d.non_existing_field
except (KeyError, AttributeError) as e:
    print("A KeyError or an AttributeError exception has been caught.")
```

If you wish to handle different exceptions in different ways, you can provide a separate exception block for each type. In this example, we still catch the `KeyError` and `AttributeError` , but handle the exceptions in different manners.

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except KeyError as e:
    print("A KeyError has occurred")
except AttributeError as e:
    print("An AttributeError has occurred.")
```

Else

Code in an else block will only be run if no exceptions were raised by the code in the `try` block. This is useful if you have some code you don't want to run if an exception is thrown, but you don't want exceptions thrown by that code to be caught.

For example:

```
try:
    data = {1: 'one', 2: 'two'}
    print(data[1])
except KeyError as e:
    print('key not found')
else:
    raise ValueError()
# Output: one
# Output: ValueError
```

Re-raising exceptions

Sometimes you want to catch an exception just to inspect it, e.g. for logging purposes. After the inspection, you want the exception to continue propagating as it did before.

In this case, simply use the `raise` statement with no parameters.

```
try:
    5 / 0
except ZeroDivisionError:
    print("Got an error")
    raise
```

Creating custom exception types

Create a class inheriting from `Exception` :

```
class FooException(Exception):
    pass
try:
    raise FooException("insert description here")
except FooException:
    print("A FooException was raised.")
```

or another exception type:

```
class NegativeError(ValueError):
      pass

def foo(x):
    # function that only accepts positive values of x
    if x < 0:
        raise NegativeError("Cannot process negative numbers")
    ...  # rest of function body
try:
    result = foo(int(input("Enter a positive integer: ")))  # raw_input in Python 2.x
except NegativeError:
```

```
    print("You entered a negative number!")
print("The result was " + str(result))
```

Exception Hierarchy

Exception handling occurs based on an exception hierarchy, determined by the inheritance structure of the exception classes.

For example, `IOError` and `OSError` are both subclasses of `EnvironmentError` . Code that catches an `IOError` will not catch an `OSError` . However, code that catches an `EnvironmentError` will catch both `IOError` s and `OSError` s.

The hierarchy of built-in exceptions:

Python 2.x ≥2.3

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
```

Python 3.x ≥3.0

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
```

```
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
```

Practical examples of exception handling

User input

Imagine you want a user to enter a number via `input`. You want to ensure that the input is a number. You can use `try` / `except` for this:

Python 3.x ≥3.0

```
while True:
    try:
        nb = int(input('Enter a number: '))
        break
    except ValueError:
        print('This is not a number, try again.')
```

Note: Python 2.x would use `raw_input` instead; the function `input` exists in Python 2.x but has different semantics. In the above example, `input` would also accept expressions such as `2 + 2` which evaluate to a number.

If the input could not be converted to an integer, a `ValueError` is raised. You can catch it with `except`. If no exception is raised, `break` jumps out of the loop. After the loop, `nb` contains an integer.

Dictionaries

Imagine you are iterating over a list of consecutive integers, like `range(n)`, and you have a list of dictionaries `d` that contains information about things to do when you encounter some particular integers, say *skip the d[i] next ones*.

```
i = 0
n = 50
d = [{7: 3}, {25: 9}, {38: 5}]
while i < n:
    do_stuff(i)
    try:
       i += d[i]
    except KeyError:
       i += 1
```

A `KeyError` will be raised when you try to get a value from a dictionary for a key that doesn't exist.

Raising Exceptions

If your code encounters a condition it doesn't know how to handle, such as an incorrect parameter, it should raise the appropriate exception.

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Did not get an even number")

    return odds + 1
```

Exceptions are Objects too

Exceptions are just regular Python objects that inherit from the built-in `BaseException`. A Python script can use the `raise` statement to interrupt execution, causing Python to print a stack trace of the call stack at that point and a representation of the exception instance. For example:

```
>>> def failing_function():
...     raise ValueError('Example error!')
>>> failing_function()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in failing_function
ValueError: Example error!
```

which says that a `ValueError` with the message `'Example error!'` was raised by our `failing_function()`, which was executed in the interpreter.

Calling code can choose to handle any and all types of exception that a call can raise:

```
>>> try:
...     failing_function()
... except ValueError:
...     print('Handled the error')
Handled the error
```

You can get hold of the exception objects by assigning them in the `except...` part of the exception handling code:

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('Caught exception', repr(e))
Caught exception ValueError('Example error!',)
```

A complete list of built-in Python exceptions along with their descriptions can be found in the Python Documentation: https://docs.python.org/3.5/library/exceptions.html . And here is the full list arranged hierarchically: Exception Hierarchy .

Running clean-up code with finally

Sometimes, you may want something to occur regardless of whatever exception happened, for example, if you have to clean up some resources.

The `finally` block of a `try` clause will happen regardless of whether any exceptions were raised.

```
resource = allocate_some_expensive_resource()
try:
    do_stuff(resource)
except SomeException as e:
    log_error(e)
    raise  # re-raise the error
finally:
    free_expensive_resource(resource)
```

This pattern is often better handled with context managers (using the `with` statement).

Chain exceptions with raise from

In the process of handling an exception, you may want to raise another exception. For example, if you get an `IOError` while reading from a file, you may want to raise an application-specific error to present to the users of your library, instead.

Python 3.x ≥3.0

You can chain exceptions to show how the handling of exceptions proceeded:

```
>>> try:
    5 / 0
except ZeroDivisionError as e:
    raise ValueError("Division failed") from e

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: Division failed
```

Syntax

```
raise exception
```

```
raise # re-raise an exception that’s already been raised
raise exception from cause # Python 3 - set exception cause
raise exception from None # Python 3 - suppress all exception context
try:
except [exception types] [ as identifier ] :
else:
finally:
```

Parameters

Remarks