5. Exceptions - Python in a Nutshell, 3rd Edition

Chapter 5. Exceptions

Python uses exceptions to communicate errors and anomalies. An *exception* is an object that indicates an error or anomaly. When Python detects an error, it *raises* an exception—that is, Python signals the occurrence of an anomalous condition by passing an exception object to the exception-propagation mechanism. Your code can explicitly raise an exception by executing a `raise` statement.

Handling an exception means receiving the exception object from the propagation mechanism and performing whatever actions are needed to deal with the anomalous situation. If a program does not handle an exception, the program terminates with an error traceback message. However, a program can handle exceptions and keep running despite errors or other abnormal conditions.

Python also uses exceptions to indicate some special situations that are not errors, and are not even abnormal. For example, as covered in "Iterators", calling the `next` built-in on an iterator raises `StopIteration` when the iterator has no more items. This is not an error; it is not even an anomaly, since most iterators run out of items eventually. The optimal strategies for checking and handling errors and other special situations in Python are therefore different from what might be best in other languages, and we cover such considerations in "Error-Checking Strategies". This chapter also covers the `logging` module of the Python standard library, in "Logging Errors", and the `assert` Python statement, in "The assert Statement".

The try Statement

The `try` statement provides Python's exception-handling mechanism. It is a compound statement that can take one of two forms:

- A `try` clause followed by one or more `except` clauses (and optionally exactly one `else` clause)
- A `try` clause followed by exactly one `finally` clause

A `try` statement can also have `except` clauses (and optionally an `else` clause) followed by a `finally` clause, as covered at "The try/except/finally Statement".

try/except

Here's the syntax for the `try`/`except` form of the `try` statement:

```
try:    statement(s)except [expression [as target]]:    statement(s)[else:
statement(s)]
```

This form of the `try` statement has one or more `except` clauses, as well as an optional `else` clause.

The body of each `except` clause is known as an *exception handler*. The code executes when the `expression` in the `except` clause matches an exception object propagating from the `try` clause. `expression` is a class (or tuple of classes, enclosed in parentheses), and matches any instance of one of those classes or any of their subclasses. The optional `target` is an identifier that names a variable that Python binds to the exception object just before the exception handler executes. A handler can also obtain the current exception object by calling the `exc_info` function

of module `sys` (covered in Table 7-3).

Here is an example of the `try`/`except` form of the `try` statement:

```
try:
    1/0
except ZeroDivisionError:
          'caught divide-by-0
    print(attempt'
)
```

If a `try` statement has several `except` clauses, the exception-propagation mechanism tests the `except` clauses in order; the first `except` clause whose expression matches the exception object executes as the handler.

Place handlers for specific exceptions before more general ones

Always place exception handlers for specific cases before handlers for more general cases: if you place a general case first, the more specific `except` clauses that follow never enter the picture.

The last `except` clause may lack an expression. This clause handles any exception that reaches it during propagation. Such unconditional handling is rare, but it does occur, generally in wrapper functions that must perform some extra task before re-raising an exception, as we'll discuss in "The raise Statement".

Avoid writing a "bare except" that doesn't re-raise the exception

Beware of using a "bare `except`" (an `except` clause without an expression) unless you're re-raising the exception in it: such sloppy style can make bugs very hard to find, since the bare `except` is over-broad and can easily mask coding errors and other kinds of bugs.

Exception propagation terminates when it finds a handler whose expression matches the exception object. When a `try` statement is nested (lexically in the source code, or dynamically within function calls) in the `try` clause of another `try` statement, a handler established by the inner `try` is reached first during propagation, and therefore handles the exception when it matches the expression. This may not be what you want. For example:

```
                                        caught an
try:    try:        1/0    except:        print('exception          ')except
                      caught divide-by-0                          caught an
ZeroDivisionError:    print('attempt                    ')# prints: exception
```

In this case, it does not matter that the handler established by the clause `except ZeroDivisionError:` in the outer `try` clause is more specific and appropriate than the catch-all `except:` in the inner `try` clause. The outer `try` does not even enter into the picture, because the exception doesn't propagate out of the inner `try`. For more on exception propagation, see "Exception Propagation".

The optional `else` clause of `try`/`except` executes only when the `try` clause terminates normally. In other words, the `else` clause does not execute when an exception propagates from the `try` clause, or when the `try` clause exits with a `break`, `continue`, or `return` statement. The handlers established by `try`/`except` cover only the `try` clause, not the `else` clause. The `else` clause is useful to avoid accidentally handling unexpected exceptions. For example:

```
                    'is        '
print(repr(value), '    , end='  )
try:
    value + 0
except TypeError:
    # not a number, maybe a
    string...?
    try:
        value + ''
    except TypeError:
              'neither a number nor a
        print(string'
)
    else:
              'some kind of
        print(string'              )
else:
          'some kind of
    print(number'              )
```

try/finally

Here's the syntax for the `try`/`finally` form of the `try` statement:

```
try:    statement(s)finally:    statement(s)
```

This form has exactly one `finally` clause (and cannot have an `else` clause—unless it also has one or more `except` clauses, as covered in "The try/except/finally Statement").

The `finally` clause establishes what is known as a *clean-up handler*. The code always executes after the `try` clause terminates in any way. When an exception propagates from the `try` clause, the `try` clause terminates, the clean-up handler executes, and the exception keeps propagating. When no exception occurs, the clean-up handler executes anyway, regardless of whether the `try` clause reaches its end or exits by executing a `break`, `continue`, or `return` statement.

Clean-up handlers established with `try/finally` offer a robust and explicit way to specify finalization code that must always execute, no matter what, to ensure consistency of program state and/or external entities (e.g., files, databases, network connections); such assured finalization, however, is usually best expressed via a *context manager* used in a `with` statement (see "The with Statement"). Here is an example of the `try`/`finally` form of the `try` statement:

```
f = open(some_file, 'w')
try:
    do_something_with_file(f
)
finally:
    f.close()
```

and here is the corresponding, more concise and readable, example of using `with` for exactly the same purpose:

```
with open(some_file, 'w') as f:
    do_something_with_file(f)
```

Avoid break and return statements in a finally clause

A `finally` clause may not directly contain a `continue` statement, but it's allowed to contain a `break` or `return` statement. Such usage, however, makes your program less clear: exception propagation stops when such a `break` or `return` executes, and most programmers would not expect propagation to be stopped within a `finally` clause. The usage may confuse people who are reading your code, so avoid it.

The try/except/finally Statement

A `try`/`except`/`finally` statement, such as:

```
try:    ...guarded clause...except ...expression...:    ...exception handler code...
finally:    ...clean-up code...
```

is equivalent to the nested statement:

```
try:    try:        ...guarded clause...    except ...expression...:        ...
exception handler code...finally:    ...clean-up code...
```

A `try` statement can have multiple `except` clauses, and optionally an `else` clause, before a terminating `finally` clause. In all variations, the effect is always as just shown—that is, just like nesting a `try`/`except` statement, with all the `except` clauses and the `else` clause if any, into a containing `try`/`finally` statement.

The with Statement and Context Managers

The `with` statement is a compound statement with the following syntax:

```
with expression [as varname]:    statement(s)
```

The semantics of `with` are equivalent to:

```
_normal_exit = True_manager = expressionvarname = _manager.__enter__()try:
statement(s)except:    _normal_exit = False    if not _manager.__exit__(*sys.exc_info(
                                    propagate if __exit__ returns a true
)):        raise    # exception does notvalue
finally:    if _normal_exit:        _manager.__exit__(None, None, None)
```

where *`_manager`* and *`_normal_exit`* are arbitrary internal names that are not used elsewhere in the current scope. If you omit the optional `as varname` part of the `with` clause, Python still calls *`_manager`*`.__enter__()`, but doesn't bind the result to any name, and still calls *`_manager`*`.__exit__()` at block termination. The object returned by the `expression`, with methods `__enter__` and `__exit__`, is known as a *context manager*.

The `with` statement is the Python embodiment of the well-known C++ idiom "resource acquisition is initialization" (RAII): you need only write context manager classes—that is, classes with two special methods `__enter__` and

`__exit__` . `__enter__` must be callable without arguments. `__exit__`must be callable with three arguments: all `None` if the body completes without propagating exceptions, and otherwise the type, value, and traceback of the exception. This provides the same guaranteed finalization behavior as typical `ctor`/`dtor` pairs have for `auto` variables in C++, and `try`/`finally` statements have in Python or Java. In addition, you gain the ability to finalize differently depending on what exception, if any, propagates, as well as optionally blocking a propagating exception by returning a true value from `__exit__`.

For example, here is a simple, purely illustrative way to ensure `<name>` and `</name>` tags are printed around some other output:

```
class tag(object):
    def __init__(self, tagname):
        self.tagname = tagname
    def __enter__(self):
        print('<{}>'.format(self.tagname), end='')
    def __exit__(self, etyp, einst, etb):
        print('</{}>'.format(self.tagname))
# to be used
as:
tt = tag('sometag')
...
with tt:
    ...statements printing output to be enclosed in
       a matched open/close `sometag` pair
```

A simpler way to build context managers is the `contextmanager` decorator in the `contextlib` module of the standard Python library, which turns a generator function into a factory of context manager objects. `contextlib` also offers wrapper function `closing` (to call some object's `close` method upon `__exit__`), and also further, more advanced utilities in v3.

The `contextlib` way to implement the `tag` context manager, having imported `contextlib` earlier, is:

```
@contextlib.contextmanager
def tag(tagname):
    print('<{}>'.format(tagname), end=''
)
    try:
        yield
    finally:
        print('</{}>'.format(tagname))
# to be used the same way as
before
```

Many more examples and detailed discussion can be found in PEP 343.

New in 3.6: `contextlib` now also has an `AbstractContextManager` class that can act as a base class for context managers.

Generators and Exceptions

To help generators cooperate with exceptions, `yield` statements are allowed inside `try`/`finally` statements.

Moreover, generator objects have two other relevant methods, `throw` and `close`. Given a generator object `g`, built by calling a generator function, the `throw` method's signature is, in v2:

```
g.throw(exc_type, exc_value=None, exc_traceback=None]
```

and, in v3, the simpler:

```
g.throw(exc_value)
```

When the generator's caller calls `g.throw`, the effect is just as if a `raise` statement with the same arguments executed at the spot of the `yield` at which generator `g` is suspended.

The generator method `close` has no arguments; when the generator's caller calls `g.close()`, the effect is just like calling `g.throw(GeneratorExit())`. `GeneratorExit` is a built-in exception class that inherits directly from `BaseException`. A generator's `close` method should re-raise (or propagate) the `GeneratorExit` exception, after performing whatever clean-up operations the generator may need. Generators also have a finalizer (special method `__del__`) that is exactly equivalent to the method `close`.

Exception Propagation

When an exception is raised, the exception-propagation mechanism takes control. The normal control flow of the program stops, and Python looks for a suitable exception handler. Python's `try` statement establishes exception handlers via its `except` clauses. The handlers deal with exceptions raised in the body of the `try` clause, as well as exceptions propagating from functions called by that code, directly or indirectly. If an exception is raised within a `try` clause that has an applicable `except` handler, the `try` clause terminates and the handler executes. When the handler finishes, execution continues with the statement after the `try` statement.

If the statement raising the exception is not within a `try` clause that has an applicable handler, the function containing the statement terminates, and the exception propagates "upward" along the stack of function calls to the statement that called the function. If the call to the terminated function is within a `try` clause that has an applicable handler, that `try` clause terminates, and the handler executes. Otherwise, the function containing the call terminates, and the propagation process repeats, *unwinding* the stack of function calls until an applicable handler is found.

If Python cannot find any applicable handler, by default the program prints an error message to the standard error stream (file *sys.stderr*). The error message includes a traceback that gives details about functions terminated during propagation. You can change Python's default error-reporting behavior by setting `sys.excepthook` (covered in Table 7-3). After error reporting, Python goes back to the interactive session, if any, or terminates if no interactive session is active. When the exception class is `SystemExit`, termination is silent, and ends the interactive session, if any.

Here are some functions to show exception propagation at work:

```
def f():
          'in f, before
    print(1/0'             )
    1/0
# raises a ZeroDivisionError
exception
          'in f, after
    print(1/0'            )

def g():
          'in g, before
    print(f()'             )
    f()
          'in g, after
    print(f()'            )

def h():
          'in h, before
    print(g()'             )
    try:
        g()
              'in h, after
        print(g()'            )
    except ZeroDivisionError:
              'ZD exception
        print(caught'              )
          'function h
    print(ends'            )
```

Calling the `h` function prints the following:

```
>>> h()in h, before g()in g, before f()in f, before 1/0ZD exception caughtfunction h
ends
```

That is, none of the “after” `print` statements execute, since the flow of exception propagation “cuts them off.”

The function `h` establishes a `try` statement and calls the function `g` within the `try` clause. `g`, in turn, calls `f`, which performs a division by `0`, raising an exception of the class `ZeroDivisionError`. The exception propagates all the way back to the `except` clause in `h`. The functions `f` and `g` terminate during the exception-propagation phase, which is why neither of their “after” messages is printed. The execution of `h`’s `try` clause also terminates during the exception-propagation phase, so its “after” message isn’t printed either. Execution continues after the handler, at the end of `h`’s `try`/`except` block.

The raise Statement

You can use the `raise` statement to raise an exception explicitly. `raise` is a simple statement with the following syntax (acceptable both in v2 and v3):

```
raise [expression]
```

Only an exception handler (or a function that a handler calls, directly or indirectly) can use `raise` without any expression. A plain `raise` statement re-raises the same exception object that the handler received. The handler terminates, and the exception propagation mechanism keeps going up the call stack, searching for other applicable handlers. Using `raise` without any expression is useful when a handler discovers that it is unable to handle an exception it receives, or can handle the exception only partially, so the exception should keep propagating to allow handlers up the call stack to perform their own handling and cleanup.

When `expression` is present, it must be an instance of a class inheriting from the built-in class `BaseException`, and Python raises that instance.

In v2 only, `expression` could also be a class object, which `raise` instantiated to raise the resulting instance, and another expression could follow to provide the argument for instantiation. We recommend that you ignore these complications, which are not present in v3 nor needed in either version: just instantiate the exception object you want to raise, and `raise` that instance.

Here's an example of a typical use of the `raise` statement:

```
def cross_product(seq1, seq2):
    if not seq1 or not seq2:
                         'Sequence arguments must be non-
        raise ValueError(empty'
)
    return [(x1, x2) for x1 in seq1 for x2 in seq2]
```

This `cross_product` example function returns a list of all pairs with one item from each of its sequence arguments, but first it tests both arguments. If either argument is empty, the function raises `ValueError` rather than just returning an empty list as the list comprehension would normally do.

Check only what you need to

There is no need for `cross_product` to check whether `seq1` and `seq2` are iterable: if either isn't, the list comprehension itself raises the appropriate exception, presumably a `TypeError`.

Once an exception is raised, by Python itself or with an explicit `raise` statement in your code, it's up to the caller to either handle it (with a suitable `try`/`except` statement) or let it propagate further up the call stack.

Don't use raise for duplicate, redundant error checks

Use the `raise` statement only to raise additional exceptions for cases that would normally be okay but that your specification defines to be errors. Do not use `raise` to duplicate the same error-checking that Python already, implicitly, does on your behalf.

Exception Objects

Exceptions are instances of subclasses of `BaseException`. Any exception has attribute `args`, the tuple of arguments used to create the instance; this error-specific information is useful for diagnostic or recovery purposes. Some exception classes interpret `args` and conveniently supply them as named attributes.

The Hierarchy of Standard Exceptions

Exceptions are instances of subclasses of `BaseException` (in v2 only, for backward compatibility, it's possible to treat some other objects as "exceptions," but we do not cover this legacy complication in this book).

The inheritance structure of exception classes is important, as it determines which `except` clauses handle which exceptions. Most exception classes extend the class `Exception`; however, the classes `KeyboardInterrupt`, `GeneratorExit`, and `SystemExit` inherit directly from `BaseException` and are not subclasses of `Exception`. Thus, a handler clause `except Exception as e:` does not catch `KeyboardInterrupt`, `GeneratorExit`, or `SystemExit` (we cover exception handlers in "try/except"). Instances of `SystemExit` are normally raised via the `exit` function in module `sys` (covered in Table 7-3). We cover `GeneratorExit` in "Generators and Exceptions". When the user hits Ctrl-C, Ctrl-Break, or other interrupting keys on their keyboard, that raises `KeyboardInterrupt`.

v2 only introduces another subclass of `Exception`: `StandardError`. Only `StopIteration` and `Warning` classes inherit directly from `Exception` (`StopIteration` is part of the iteration protocol, covered in "Iterators"; `Warning` is covered in "The warnings Module")—all other standard exceptions inherit from `StandardError`. However, v3 has removed this complication. So, in v2, the inheritance hierarchy from `BaseException` down is, roughly:

```
BaseException
  Exception
    StandardError
      AssertionError, AttributeError, BufferError, EOFError,
      ImportError, MemoryError, SystemError, TypeError
      ArithmeticError
        FloatingPointError, OverflowError,
ZeroDivisionError
      EnvironmentError
        IOError, OSError
      LookupError
        IndexError, KeyError
      NameError
        UnboundLocalError
      RuntimeError
        NotImplementedError
      SyntaxError
        IndentationError
      ValueError
        UnicodeError
          UnicodeDecodeError, UnicodeEncodeError
    StopIteration
    Warning
  GeneratorExit
  KeyboardInterrupt
  SystemExit
```

There are other exception subclasses (in particular, `Warning` has many), but this is the gist of the hierarchy—in v2. In v3, it's simpler, as `StandardError` disappears (and all of its direct subclasses become direct subclasses of `Exception`), and so do `EnvironmentError` and `IOError` (the latter becomes a synonym of `OSError`, which, in v3, directly subclasses `Exception`), as per the following condensed table:

```
BaseException  Exception    AssertionError    ...     OSError
# no longer a subclass of
EnvironmentError                              RuntimeError    StopIteration
SyntaxError    ValueError    Warning     GeneratorExit  KeyboardInterrupt  SystemExit
```

Three subclasses of `Exception` are never instantiated directly. Their only purpose is to make it easier for you to specify `except` clauses that handle a broad range of related errors. These three "abstract" subclasses of `Exception` are:

`ArithmeticError`

> The base class for exceptions due to arithmetic errors (i.e., `OverflowError`, `ZeroDivisionError`, `FloatingPointError`)

`EnvironmentError`

> The base class for exceptions due to external causes (i.e., `IOError`, `OSError`, `WindowsError`), in v2 only

`LookupError`

> The base class for exceptions that a container raises when it receives an invalid key or index (i.e., `IndexError`, `KeyError`)

Standard Exception Classes

Common runtime errors raise exceptions of the following classes:

`AssertionError`

> An assert statement failed.

`AttributeError`

> An attribute reference or assignment failed.

`FloatingPointError`

> A floating-point operation failed. Extends `ArithmeticError`.

`IOError`

> An I/O operation failed (e.g., the disk was full, a file was not found, or needed permissions were missing). In v2, extends `EnvironmentError`; in v3, it's a synonym of `OSError`.

`ImportError`

> An `import` statement (covered in "The import Statement") couldn't find the module to import or couldn't find a name to be imported from the module.

`IndentationError`

> The parser encountered a syntax error due to incorrect indentation. Extends `SyntaxError`.

`IndexError`

> An integer used to index a sequence is out of range (using a noninteger as a sequence index raises

`TypeError`). Extends `LookupError`.

`KeyError`

A key used to index a mapping is not in the mapping. Extends `LookupError`.

`KeyboardInterrupt`

The user pressed the interrupt key combination (Ctrl-C, Ctrl-Break, Delete, or others, depending on the platform's handling of the keyboard).

`MemoryError`

An operation ran out of memory.

`NameError`

A variable was referenced, but its name was not bound.

`NotImplementedError`

Raised by abstract base classes to indicate that a concrete subclass must override a method.

`OSError`

Raised by functions in the module `os` (covered in "The os Module" and "Running Other Programs with the os Module") to indicate platform-dependent errors. In v2, extends `EnvironmentError`. In v3 only, it has many subclasses, covered at "OSError and subclasses (v3 only)".

`OverflowError`

The result of an operation on an integer is too large to fit into an integer. Extends `ArithmeticError`.

OverflowError still exists only for legacy/compatibility

This error never happens in modern versions of Python: integer results too large to fit into a platform's integers implicitly become long integers, without raising exceptions (indeed, in v3, there's no `long` type, just `int` for all integers). This standard exception remains a built-in for backward compatibility (to support old code that raises or tries to catch it). Do *not* use it in any new code.

`SyntaxError`

The parser encountered a syntax error.

`SystemError`

An internal error within Python itself or some extension module. You should report this to the authors and maintainers of Python, or of the extension in question, with all possible details to allow them to reproduce the problem.

`TypeError`

An operation or function was applied to an object of an inappropriate type.

`UnboundLocalError`

A reference was made to a local variable, but no value is currently bound to that local variable. Extends

`NameError`.

`UnicodeError`

An error occurred while converting Unicode to a byte string or vice versa.

`ValueError`

An operation or function was applied to an object that has a correct type but an inappropriate value, and nothing more specific (e.g., `KeyError`) applies.

`WindowsError`

Raised by functions in the module `os` (covered in "The os Module" and "Running Other Programs with the os Module") to indicate Windows-specific errors. Extends `OSError`.

`ZeroDivisionError`

A divisor (the righthand operand of a `/`, `//`, or `%` operator, or the second argument to the built-in function `divmod`) is `0`. Extends `ArithmeticError`.

OSError and subclasses (v3 only)

In v3, `OSError` subsumes many errors that v2 kept separate, such as `IOError` and `socket.error`. To compensate (and more!) in v3, `OSError` has spawned many useful subclasses, which you can catch to handle environment errors much more elegantly—see the online docs.

For example, consider this task: try to read and return the contents of a certain file; return a default string if the file does not exist; propagate any other exception that makes the file unreadable (except for the file not existing). A v2/v3 portable version:

```
import errno

def read_or_default(filepath, default):
    try:
        with open(filepath) as f:
            return f.read()
    except IOError as e:
        if e.errno == errno.ENOENT:
            return default
        else:
            raise
```

However, see how much simpler it can be in v3, using an `OSError` subclass:

```
def read_or_default(filepath, default):
    try:
        with open(filepath) as f:
            return f.read()
    except FileNotFoundError:
        return default
```

The `FileNotFoundError` subclass of `OSError`, in v3 only, makes this kind of common task much simpler and

more direct for you to express in your code.

Exceptions “wrapping” other exceptions or tracebacks

Sometimes, you incur an exception while trying to handle another. In v2, there’s not much you can do about it, except perhaps by coding and specially handling custom exception classes, as shown in the next section. v3 offers much more help. In v3, each exception instance holds its own traceback object; you can make another exception instance with a different traceback with the `with_traceback` method. Moreover, v3 automatically remembers which exception it’s handling as the “context” of any one raised during the handling.

For example, consider the deliberately broken code:

```
try: 1/0
except ZeroDivisionError:
    1+'x'
```

In v2, that code displays something like:

```
Traceback (most recent call last):  File "<stdin>", line 3, in <module>TypeError:
unsupported operand type(s) for +: 'int' and 'str'
```

which hides the zero-division error that was being handled. In v3, by contrast:

```
Traceback (most recent call last):  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zeroDuring handling of the above exception, another
exception occurred:Traceback (most recent call last):  File "<stdin>", line 3, in <
module>TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

...both exceptions, the original and the intervening one, are clearly displayed.

If this isn’t enough, in v3, you can also `raise e from ex`, with both `e` and `ex` being exception objects: `e` is the one that propagates, and `ex` is its “cause.” For all details and motivations, see PEP 3134.

Custom Exception Classes

You can extend any of the standard exception classes in order to define your own exception class. Often, such a subclass adds nothing more than a docstring:

```
class InvalidAttribute(AttributeError):

"""Used to indicate attributes that could never be
valid"""
```

Any empty class or function should have a docstring, not pass

As covered in “The pass Statement”, you don’t need a `pass` statement to make up the body of a class. The docstring (which you should always write, to document the class’s purpose) is enough to keep Python happy. Best practice for all “empty” classes (regardless of whether they are exception classes), just like for all “empty” functions,

is to always have a docstring, and no `pass` statement.

Given the semantics of `try`/`except`, raising a custom exception class such as `InvalidAttribute` is almost the same as raising its standard exception superclass, `AttributeError`. Any `except` clause that can handle `AttributeError` can handle `InvalidAttribute` just as well. In addition, client code that knows about your `InvalidAttribute` custom exception class can handle it specifically, without having to handle all other cases of `AttributeError` when it is not prepared for those. For example:

```
class SomeFunkyClass(object):
    """much hypothetical functionality
    snipped"""
    def __getattr__(self, name):
        """only clarifies the kind of attribute
        error"""
        if name.startswith('_'):
                                    'Unknown private attribute
            raise InvalidAttribute, '                           +
name
        else:
                                  'Unknown attribute
            raise AttributeError, '                   +name
```

Now client code can be more selective in its handlers. For example:

```
s = SomeFunkyClass()
try:
    value = getattr(s, thename)
except InvalidAttribute, err:
    warnings.warn(str(err))
    value = None
# other cases of AttributeError just propagate, as they're
unexpected
```

Define and raise custom exception classes

It's an excellent idea to define, and raise, custom exception classes in your modules, rather than plain standard exceptions: by using custom exceptions, you make it easier for callers of your module's code to handle exceptions that come from your module separately from others.

A special case of custom exception class that you may find useful (at least in v2, which does not directly support an exception wrapping another) is one that wraps another exception and adds information. To gather information about a pending exception, you can use the `exc_info` function from module `sys` (covered in Table 7-3). Given this, your custom exception class could be defined as follows:

```
import sys
class CustomException(Exception):

"""Wrap arbitrary pending exception, if any,
       in addition to other
info."""
    def __init__(self, *args):
        Exception.__init__(self, *args)
        self.wrapped_exc = sys.exc_info()
```

You would then typically use this class in a wrapper function such as:

```
def call_wrapped(callable, *args, **kwds):
    try:
        return callable(*args, **kwds)
    except:
                               'Wrapped function
        raise CustomException('

'propagated
exception'            )
```

Custom Exceptions and Multiple Inheritance

A particularly effective approach to custom exceptions is to multiply inherit exception classes from your module's special custom exception class and a standard exception class, as in the following snippet:

```
class CustomAttributeError(CustomException, AttributeError):
    """An AttributeError which is ALSO a
    CustomException."""
```

Now, when your code raises an instance of `CustomAttributeError`, that exception can be caught by calling code that's designed to catch all cases of `AttributeError` as well as by code that's designed to catch all exceptions raised only by your module.

Use the multiple inheritance approach for custom exceptions

Whenever you must decide whether to raise a specific standard exception, such as `AttributeError`, or a custom exception class you define in your module, consider this multiple-inheritance approach, which gives you the best of both worlds. Make sure you clearly document this aspect of your module; since the technique is not widely used, users of your module may not expect it unless you clearly and explicitly document what you are doing.

Other Exceptions Used in the Standard Library

Many modules in Python's standard library define their own exception classes, which are equivalent to the custom exception classes that your own modules can define. Typically, all functions in such standard library modules may raise exceptions of such classes, in addition to exceptions in the standard hierarchy covered in "Standard Exception Classes". For example, in v2, module `socket` supplies class `socket.error`, which is directly derived from built-in class `Exception`, and several subclasses of `error` named `sslerror`, `timeout`, `gaierror`, and `herror`; all

functions and methods in module `socket`, besides standard exceptions, may raise exceptions of class `socket.error` and subclasses thereof. We cover the main cases of such exception classes throughout the rest of this book, in chapters covering the standard library modules that supply them.

Error-Checking Strategies

Most programming languages that support exceptions raise exceptions only in rare cases. Python's emphasis is different. Python deems exceptions appropriate whenever they make a program simpler and more robust, even if that makes exceptions rather frequent.

LBYL Versus EAFP

A common idiom in other languages, sometimes known as "Look Before You Leap" (LBYL), is to check in advance, before attempting an operation, for anything that might make the operation invalid. This approach is not ideal for several reasons:

- The checks may diminish the readability and clarity of the common, mainstream cases where everything is okay.
- The work needed for checking may duplicate a substantial part of the work done in the operation itself.
- The programmer might easily err by omitting some needed check.
- The situation might change between the moment you perform the checks and the moment you attempt the operation.

The preferred idiom in Python is generally to attempt the operation in a `try` clause and handle the exceptions that may result in `except` clauses. This idiom is known as "it's Easier to Ask Forgiveness than Permission" (EAFP), a motto widely credited to Rear Admiral Grace Murray Hopper, co-inventor of COBOL. EAFP shares none of the defects of LBYL. Here is a function written using the LBYL idiom:

```
def safe_divide_1(x, y):
    if y==0:
            'Divide-by-0 attempt
        print(detected'
)
        return None
    else:
        return x/y
```

With LBYL, the checks come first, and the mainstream case is somewhat hidden at the end of the function. Here is the equivalent function written using the EAFP idiom:

```
def safe_divide_2(x, y):
    try:
        return x/y
    except ZeroDivisionError:
            'Divide-by-0 attempt
        print(detected'
)
        return None
```

With EAFP, the mainstream case is up front in a `try` clause, and the anomalies are handled in an `except` clause that lexically follows.

Proper usage of EAFP

EAFP is a good error-handling strategy, but it is not a panacea. In particular, don't cast too wide a net, catching errors that you did not expect and therefore did not mean to catch. The following is a typical case of such a risk (we cover built-in function `getattr` in Table 7-2):

```
def trycalling(obj, attrib, default, *args, **kwds):
    try:
        return getattr(obj, attrib)(*args, **kwds)
    except AttributeError:
        return default
```

The intention of function `trycalling` is to try calling a method named `attrib` on object `obj`, but to return `default` if `obj` has no method thus named. However, the function as coded does not do *just* that: it also mistakenly hides any error case where `AttributeError` is raised inside the sought-after method, silently returning `default` in those cases. This may easily hide bugs in other code. To do exactly what's intended, the function must take a little bit more care:

```
def trycalling(obj, attrib, default, *args, **kwds):
    try:
        method = getattr(obj, attrib)
    except AttributeError:
        return default
    else:
        return method(*args, **kwds)
```

This implementation of `trycalling` separates the `getattr` call, placed in the `try` clause and therefore guarded by the handler in the `except` clause, from the call of the method, placed in the `else` clause and therefore free to propagate any exceptions. The proper approach to EAFP involves frequent use of the `else` clause on `try`/`except` statements (which is more explicit, and thus better style, than just placing the nonguarded code after the whole `try`/`except` statement).

Handling Errors in Large Programs

In large programs, it is especially easy to err by making your `try`/`except` statements too wide, particularly once you have convinced yourself of the power of EAFP as a general error-checking strategy. A `try`/`except` combination is too wide when it catches too many different errors, or an error that can occur in too many different places. The latter is a problem when you need to distinguish exactly what went wrong, and where, and the information in the traceback is not sufficient to pinpoint such details (or you discard some or all of the information in the traceback). For effective error handling, you have to keep a clear distinction between errors and anomalies that you expect (and thus know how to handle) and unexpected errors and anomalies that indicate a bug in your program.

Some errors and anomalies are not really erroneous, and perhaps not even all that anomalous: they are just special cases, perhaps somewhat rare but nevertheless quite expected, which you choose to handle via EAFP rather than via LBYL to avoid LBYL's many intrinsic defects. In such cases, you should just handle the anomaly, often without even logging or reporting it.

Keep your try/except constructs narrow

Be very careful to keep `try`/`except` constructs as narrow as feasible. Use a small `try` clause that contains a small amount of code that doesn't call too many other functions, and use very specific exception-class tuples in the `except` clauses; if need be, further analyze the details of the exception in your handler code, and `raise` again as soon as you know it's not a case this handler can deal with.

Errors and anomalies that depend on user input or other external conditions not under your control are always expected, to some extent, precisely because you have no control over their underlying causes. In such cases, you should concentrate your effort on handling the anomaly gracefully, reporting and logging its exact nature and details, and keeping your program running with undamaged internal and persistent state. The breadth of `try`/`except` clauses under such circumstances should also be reasonably narrow, although this is not quite as crucial as when you use EAFP to structure your handling of not-really-erroneous special cases.

Lastly, entirely unexpected errors and anomalies indicate bugs in your program's design or coding. In most cases, the best strategy regarding such errors is to avoid `try`/`except` and just let the program terminate with error and traceback messages. (You might want to log such information and/or display it more suitably with an application-specific hook in `sys.excepthook`, as we'll discuss shortly.) In the unlikely case that your program must keep running at all costs, even under the direst circumstances, `try`/`except` statements that are quite wide may be appropriate, with the `try` clause guarding function calls that exercise vast swaths of program functionality, and broad `except` clauses.

In the case of a long-running program, make sure to log all details of the anomaly or error to some persistent place for later study (and also report some indication of the problem, so that you know such later study is necessary). The key is making sure that you can revert the program's persistent state to some undamaged, internally consistent point. The techniques that enable long-running programs to survive some of their own bugs, as well as environmental adversities, are known as checkpointing (basically, periodically saving program state, and writing the program so it can reload the saved state and continue from there) and transaction processing, but we do not cover them further in this book.

Logging Errors

When Python propagates an exception all the way to the top of the stack without finding an applicable handler, the interpreter normally prints an error traceback to the standard error stream of the process (`sys.stderr`) before terminating the program. You can rebind `sys.stderr` to any file-like object usable for output in order to divert this information to a destination more suitable for your purposes.

When you want to change the amount and kind of information output on such occasions, rebinding `sys.stderr` is not sufficient. In such cases, you can assign your own function to `sys.excepthook`: Python calls it when terminating the program due to an unhandled exception. In your exception-reporting function, output whatever information you think will help you diagnose and debug the problem and direct that information to whatever destinations you please. For example, you might use module `traceback` (covered in "The traceback Module") to help you format stack traces. When your exception-reporting function terminates, so does your program.

The logging package

The Python standard library offers the rich and powerful `logging` package to let you organize the logging of messages from your applications in systematic and flexible ways. You might write a whole hierarchy of `Logger` classes and subclasses. You might couple the loggers with instances of `Handler` (and subclasses thereof). You might also insert instances of class `Filter` to fine-tune criteria determining what messages get logged in which ways. The messages that do get emitted are formatted by instances of the `Formatter` class—indeed, the

messages themselves are instances of the `LogRecord` class. The `logging` package even includes a dynamic configuration facility, whereby you may dynamically set logging-configuration files by reading them from disk files, or even by receiving them on a dedicated socket in a specialized thread.

While the `logging` package sports a frighteningly complex and powerful architecture, suitable for implementing highly sophisticated logging strategies and policies that may be needed in vast and complicated programming systems, in most applications you may get away with using a tiny subset of the package through some simple functions supplied by the `logging` module itself. First of all, `import logging`. Then, emit your message by passing it as a string to any of the functions `debug`, `info`, `warning`, `error`, or `critical`, in increasing order of severity. If the string you pass contains format specifiers such as `%s` (as covered in "Legacy String Formatting with `%`") then, after the string, pass as further arguments all the values to be formatted in that string. For example, don't call:

```
logging.debug('foo is %r' % foo)
```

which performs the formatting operation whether it's needed or not; rather, call:

```
logging.debug('foo is %r', foo)
```

which performs formatting if and only if needed (i.e., if and only if calling `debug` is going to result in logging output, depending on the current threshold level).

Unfortunately, the `logging` module does not support the more readable formatting approach covered in "String Formatting", but only the antiquated one covered in "Legacy String Formatting with `%`". Fortunately, it's very rare that you'll need any formatting specifier, except the simple `%s` and `%r`, for logging purposes.

By default, the threshold level is `WARNING`, meaning that any of the functions `warning`, `error`, or `critical` results in logging output, but the functions `debug` and `info` don't. To change the threshold level at any time, call `logging.getLogger().setLevel`, passing as the only argument one of the corresponding constants supplied by module `logging`: `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`. For example, once you call:

```
logging.getLogger().setLevel(logging.DEBUG)
```

all of the logging functions from `debug` to `critical` result in logging output until you change level again; if later you call:

```
logging.getLogger().setLevel(logging.ERROR)
```

then only the functions `error` and `critical` result in logging output (`debug`, `info`, and `warning` won't result in logging output); this condition, too, persists only until you change level again, and so forth.

By default, logging output is to your process's standard error stream (`sys.stderr`, as covered in Table 7-3) and uses a rather simplistic format (for example, it does not include a timestamp on each line it outputs). You can control these settings by instantiating an appropriate handler instance, with a suitable formatter instance, and creating and setting a new logger instance to hold it. In the simple, common case in which you just want to set these logging parameters once and for all, after which they persist throughout the run of your program, the simplest approach is to call the `logging.basicConfig` function, which lets you set up things quite simply via named parameters. Only

the very first call to `logging.basicConfig` has any effect, and only if you call it before any of the logging functions (`debug`, `info`, and so on). Therefore, the most common use is to call `logging.basicConfig` at the very start of your program. For example, a common idiom at the start of a program is something like:

```
import logging
logging.basicConfig(
    format='%(asctime)s %(levelname)8s %(message)s'
,
    filename='/tmp/logfile.txt', filemode='w')
```

This setting emits all logging messages to a file and formats them nicely with a precise human-readable timestamp, followed by the severity level right-aligned in an eight-character field, followed by the message proper.

For excruciatingly large amounts of detailed information on the logging package and all the wonders you can perform with it, be sure to consult Python's *rich online information about it*.

The assert Statement

The `assert` statement allows you to introduce "sanity checks" into a program. `assert` is a simple statement with the following syntax:

```
assert condition[,expression]
```

When you run Python with the optimize flag (`-O`, as covered in "Command-Line Syntax and Options"), `assert` is a null operation: the compiler generates no code for it. Otherwise, `assert` evaluates `condition`. When `condition` is satisfied, `assert` does nothing. When `condition` is not satisfied, `assert` instantiates `AssertionError` with `expression` as the argument (or without arguments, if there is no `expression`) and raises the resulting instance.

`assert` statements can be an effective way to document your program. When you want to state that a significant, nonobvious condition `C` is known to hold at a certain point in a program's execution (known as an *invariant* of your program), `assert C` is often better than a comment that just states that `C` holds.

The advantage of `assert` is that, when `C` does *not* in fact hold, `assert` immediately alerts you to the problem by raising `AssertionError`, if the program is running without the `-O` flag. Once the code is thoroughly debugged, run it with `-O`, turning `assert` into a null operation and incurring no overhead (the `assert` remains in your source code to document the invariant).

Don't overuse assert

Never use `assert` for other purposes besides sanity-checking program invariants. A serious but very common mistake is to use `assert` about the values of inputs or arguments: checking for erroneous arguments or inputs is best done more explicitly, and in particular must not be turned into a null operation by a command-line flag.

The __debug__ Built-in Variable

When you run Python without option `-O`, the `__debug__` built-in variable is `True`. When you run Python with option `-O`, `__debug__` is `False`. Also, with option `-O`, the compiler generates no code for any `if` statement whose condition is `__debug__`.

To exploit this optimization, surround the definitions of functions that you call only in `assert` statements with

`if __debug__:`. This technique makes compiled code smaller and faster when Python is run with `-O`, and enhances program clarity by showing that those functions exist only to perform sanity checks.

New in 3.6: `ModuleNotFoundError` is a new, explicit subclass of `ImportError`.

Some third-party frameworks, such as pytest, materially improve the usefulness of the `assert` statement.