Chapter 6. Exceptions

Python uses *exceptions* to indicate errors and anomalies. 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 catching the exception object from the propagation mechanism and taking actions as needed to deal with the anomalous situation. If a program does not handle an exception, the program terminates with an error message and traceback message. However, a program can handle exceptions and keep running, despite errors or other anomalies, by using the **`try`** statement with **`except`** clauses.

Python also uses exceptions to indicate some situations that are not errors, and not even abnormal. For example, as covered in **"Iterators"**, calling the `next` built-in function 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 those in other languages; we cover them in **"Error-Checking Strategies"**.

This chapter shows how to use exceptions for errors and special situations. It also covers the `logging` module of the standard library, in **"Logging Errors"**, and the **`assert`** statement, in **"The assert Statement"**.

The try Statement

The **try** statement is Python's core exception handling mechanism. It's a compound statement with three kinds of optional clauses:

1. It may have zero or more **except** clauses, defining how to handle particular classes of exceptions.
2. If it has **except** clauses, then it may also have, right afterwards, one **else** clause, executed only if the **try** suite raised no exceptions.
3. Whether or not it has **except** clauses, it may have a single **finally** clause, unconditionally executed, with the behavior covered in **"try/except/finally"**.

Python's syntax requires the presence of at least one **except** clause or a **finally** clause, both of which might also be present in the same statement; **else** is only valid following one or more **except**s.

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)]
[finally:
    statement(s)]
```

This form of the **try** statement has one or more **except** clauses, as well as an optional **else** clause (and an optional **finally** clause, whose meaning does not depend on whether **except** and **else** clauses are present: we cover this in the following section).

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, in parentheses, and matches any instance of one of those classes or 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 (3.11+ or the `exception` function) of the module `sys` (covered in **Table 9-3**).

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

```
try:
    1/0
    print('not executed')
except ZeroDivisionError:
    print('caught divide-by-0 attempt')
```

When an exception is raised, execution of the **try** suite immediately ceases. If a **try** statement has several **except** clauses, the exception propagation mechanism checks the **except** clauses in order; the first **except** clause whose expression matches the exception object executes as the handler, and the exception propagation mechanism checks no further **except** clauses after that.

SPECIFIC BEFORE GENERAL

Place handlers for specific cases before handlers for more general cases: when you place a general case first, the more specific **except** clauses that follow never execute.

The last **except** clause need not specify an expression. An **except** clause without any expression handles any exception that reaches it during propagation. Such unconditional handling is rare, but it does occur, often

in “wrapper” functions that must perform some extra task before re-raising an exception (see **“The raise Statement”**).

AVOID A “BARE EXCEPT” THAT DOESN’T RE-RAISE

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 overly broad and can easily mask coding errors and other kinds of bugs by allowing execution to continue after an unanticipated exception.

New programmers who are “just trying to get things to work” may even write code like:

```
try:
    # ...code that has a problem...
except:
    pass
```

This is a dangerous practice, since it catches important process-exiting exceptions such as `KeyboardInterrupt` or `SystemExit`—a loop with such an exception handler can’t be exited with Ctrl-C, and possibly not even terminated with a system **kill** command. At the very least, such code should use **except** `Exception:`, which is still overly broad but at least does not catch the process-exiting exceptions.

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 on propagation, so it handles the exception when it matches it. This may not be what you want. Consider this example:

```
try:
    try:
        1/0
    except:
```

```
        print('caught an exception')
except ZeroDivisionError:
    print('caught divide-by-0 attempt')
# prints: caught an 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 than the catch-all **except**: in the inner **try** clause. The outer **try** does not enter into the picture: 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. 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:

```
print(repr(value), 'is ', end=' ')
try:
    value + 0
except TypeError:
    # not a number, maybe a string...?
    try:
        value + ''
    except TypeError:
        print('neither a number nor a string')
    else:
        print('some kind of string')
else:
    print('some kind of 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 one **`finally`** clause, and no `else` clause (unless it also has one or more **`except`** clauses, as covered in the following section).

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

Cleanup 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 is nowadays usually best expressed via a *context manager* used in a **`with`** statement (see **"The with Statement and Context Managers"**). 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 contain one or more of the statements **`continue`**, 3.8+ **`break`**, or **`return`**. However, such usage may make your program less clear: exception propagation stops when such a statement executes, and most programmers would not expect propagation to be stopped within a **`finally`** clause. This usage may confuse people who are reading your code, so we recommend you avoid it.

---

try/except/finally

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

```
try:
    ...guarded clause...
except ...expression...:
    ...exception handler code...
finally:
    ...cleanup code...
```

is equivalent to the nested statement:

```
try:
    try:
        ...guarded clause...
    except ...expression...:
        ...exception handler code...
finally:
    ...cleanup 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, it's just like nesting a **`try/except`** state-

ment, with all the **except** clauses and the **else** clause, if any, into a containing **try/finally** statement.

The raise Statement

You can use the **raise** statement to raise an exception explicitly. **raise** is a simple statement with the following syntax:

```
raise [expression [from exception]]
```

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.

When **from** *exception* is included (which can only occur in an **except** block that receives *exception*), Python raises the received expression "nested" in the newly raised exception expression. **"Exceptions "wrapping" other exceptions or tracebacks"** describes this in more detail.

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

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

❶ Some people consider raising a standard exception here to be inappropriate, and would prefer to raise an instance of a custom exception, as covered later in this chapter; this book's authors disagree with this opinion.

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 is 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 REDUNDANT ERROR CHECKS

Use the `raise` statement only to raise additional exceptions for cases that would normally be OK 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.

The with Statement and Context Managers

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

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

# 3.10+ multiple context managers for a with statement
# can be enclosed in parentheses
with (expression [as varname], ...):
    statement(s)
```

The semantics of **with** are equivalent to:

```
_normal_exit = True
_manager = expression
varname = _manager.__enter__()
try:
    statement(s)
except:
    _normal_exit = False
    if not _manager.__exit_(*sys.exc_info()):
        raise
    # note that exception does not propagate if __exit__ returns
    # a true value
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** when 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, they can finalize differently depending on what exception, if any, propagates, and optionally block 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 (note that context manager classes often have lowercase names, rather than following the normal title case convention for class names):

```
class enclosing_tag:
    def __init__(self, tagname):
        self.tagname = tagname
    def __enter__(self):
        print(f'<{self.tagname}>', end='')
    def __exit__(self, etyp, einst, etb):
        print(f'</{self.tagname}>')

# to be used as:
with enclosing_tag('sometag'):
    # ...statements printing output to be enclosed in
    # a matched open/close `sometag` pair...
```

A simpler way to build context managers is to use the `contextmanager` decorator in the `contextlib` module of the Python standard library. This decorator turns a generator function into a factory of context manager objects.

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

```
@contextlib.contextmanager
def enclosing_tag(tagname):
```

```
    print(f'<{tagname}>', end='')
    try:
        yield
    finally:
        print(f'</{tagname}>')
# to be used the same way as before
```

`contextlib` supplies, among others, the class and functions listed in Table 6-1.

Table 6-1. Commonly used classes and functions in the `contextlib` module

`AbstractContextManager`	`AbstractContextManager` An abstract base class with two overridable `__enter__`, which defaults to **`return`** `self`, a which defaults to **`return`** `None`.

`chdir`	`chdir(`*`dir_path`*`)`
	3.11+ A context manager whose `__enter__`
	current working directory path and perfor
	`os.chdir(`*`dir_path`*`)`, and whose `__exit__`
	`os.chdir(`*`saved_path`*`)`.

`closing`	`closing(`*`something`*`)` A context manager whose `__enter__` metho *`something`*, and whose `__exit__` method ca *`something`*`.close()`.
`contextmanager`	`contextmanager` A decorator that you apply to a generator to context manager.
`nullcontext`	`nullcontext(`*`something`*`)` A context manager whose `__enter__` metho *`something`*, and whose `__exit__` method do
`redirect_stderr`	`redirect_stderr(`*`destination`*`)` A context manager that temporarily redire

	body of the **`with`** statement, `sys.stderr` to t object *`destination`*.
`redirect_stdout`	`redirect_stdout(`*`destination`*`)` A context manager that temporarily redire body of the **`with`** statement, `sys.stdout` to t object *`destination`*.

`suppress`	`suppress(*`*`exception_classes`*`)`
	A context manager that silently suppresses
	occurring in the body of the **`with`** statement
	classes listed in *`exception_classes`*. For ins
	function to delete a file ignores `FileNotFou`
	
	`def delete_file(filename):`
	`  with contextlib.suppress(FileNo`
	`  os.remove(filename)`
	
	Use sparingly, since silently suppressing ex
	bad practice.

For more details, examples, “recipes,” and even more (somewhat abstruse) classes, see Python’s **online docs**.

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:

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

When the generator's caller calls `g.throw`, the effect is just as if a **`raise`** statement with the same argument 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 like calling `g.throw(GeneratorExit())`.[1] `GeneratorExit` is a built-in exception class that inherits directly from `BaseException`. Generators also have a finalizer (the special method `__del__`) that implicitly calls `close` when the generator object is garbage-collected.

If a generator raises or propagates a `StopIteration` exception, Python turns the exception's type into `RuntimeError`.

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 (in the absence of any explicit change to the flow of control, such as a **`raise`** or **`return`** 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 (`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 8-3**). After error reporting, Python goes back to the interactive session, if any, or terminates if execution was not interactive. When the exception type is `SystemExit`, termination is silent and ends the interactive session, if any.

Here are some functions to show exception propagation at work:

```
def f():
    print('in f, before 1/0')
    1/0    # raises a ZeroDivisionError exception
    print('in f, after 1/0')
def g():
    print('in g, before f()')
    f()
    print('in g, after f()')
def h():
    print('in h, before g()')
    try:
        g()
        print('in h, after g()')
    except ZeroDivisionError:
        print('ZD exception caught')
    print('function h ends')
```

Calling the `h` function prints the following:

```
in h, before g()
in g, before f()
in f, before 1/0
ZD exception caught
function 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 type `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.

Exception Objects

Exceptions are instances of `BaseException` (more specifically, instances of one of its subclasses). **Table 6-2** lists the attributes and methods of `BaseException`.

Table 6-2. Attributes and methods of the `BaseException` class

`__cause__`	*`exc`*`.__cause__` Returns the parent exception of an exception raised using **`raise from`**.
`__notes__`	*`exc`*`.__notes__` **3.11+** Returns a list of `str`s added to the exception using `add_note`. This attribute only exists if `add_note` has been called at least once, so the safe way to access this list is with `getattr(`*`exc`*`, '__notes__', [])`.
`add_note`	*`exc`*`.add_note(`*`note`*`)` **3.11+** Appends the `str` *`note`* to the notes on this exception. These notes are shown after the traceback when displaying the exception.

`args`	*`exc`*`.args` Returns a tuple of the arguments used to construct the exception. This error-specific information is useful for diagnostic or recovery purposes. Some exception classes interpret `args` and set convenient named attributes on the classes' instances.
`with_traceback`	*`exc`*`.with_traceback(`*`tb`*`)` Returns a new exception, replacing the original exception's traceback with the new traceback *`tb`*, or with no traceback if *`tb`* is **`None`**. Can be used to trim the original traceback to remove internal library function call frames.

The Hierarchy of Standard Exceptions

As mentioned previously, exceptions are instances of subclasses of `BaseException`. 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 covered exception handlers in **"try/except"** and `GeneratorExit` in **"Generators and Exceptions"**). Instances of `SystemExit` are normally raised via the `exit` function in the `sys` module (covered in **Table 8-3**). When the user hits Ctrl-C, Ctrl-Break, or other interrupting keys on their keyboard, that raises `KeyboardInterrupt`.

The hierarchy of built-in exception classes is, roughly:

```
BaseException
  Exception
    AssertionError, AttributeError, BufferError, EOFError,
```

```
      MemoryError, ReferenceError, OsError, StopAsyncIteration,
      StopIteration, SystemError, TypeError
      ArithmeticError (abstract)
        OverflowError, ZeroDivisionError
      ImportError
        ModuleNotFoundError, ZipImportError
      LookupError (abstract)
        IndexError, KeyError
      NameError
        UnboundLocalError
      OSError
        ...
      RuntimeError
        RecursionError
        NotImplementedError
      SyntaxError
        IndentationError
          TabError
      ValueError
        UnsupportedOperation
        UnicodeError
          UnicodeDecodeError, UnicodeEncodeError,
          UnicodeTranslateError
      Warning
        ...
    GeneratorExit
    KeyboardInterrupt
    SystemExit
```

There are other exception subclasses (in particular, `Warning` and `OSError` have many, whose omission is indicated here with ellipses), but this is the gist. A complete list is available in Python's **online docs**.

The classes marked "(abstract)" are never instantiated directly; their purpose is to make it easier for you to specify **`except`** clauses that handle a range of related errors.

Standard Exception Classes

Table 6-3 lists exception classes raised by common runtime errors.

Table 6-3. Standard exception classes

Exception class	Raised when
`AssertionError`	An **`assert`** statement failed.
`AttributeError`	An attribute reference or assignment failed.
`ImportError`	An **`import`** or **`from...import`** statement (covered in **"The import Statement"**) couldn't find the module to import (in this case, what Python raises is actually an instance of `ImportError`'s subclass `ModuleNotFoundError`), or couldn't find a name to be imported from the module.
`IndentationError`	The parser encountered a syntax error due to incorrect indentation. Subclasses `SyntaxError`.
`IndexError`	An integer used to index a sequence is out of range (using a noninteger as a sequence index raises `TypeError`). Subclasses `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).
`KeyError`	A key used to index a mapping is not in the mapping. Subclasses `LookupError`.
`MemoryError`	An operation ran out of memory.
`NameError`	A name was referenced, but it was not bound to any variable in the current scope.

Exception class	Raised when
`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. `OSError` has many subclasses, covered in the following subsection.
`RecursionError`	Python detected that the recursion depth has been exceeded. Subclasses `RuntimeError`.
`RuntimeError`	Raised for any error or anomaly not otherwise classified.
`SyntaxError`	Python's parser encountered a syntax error.
`SystemError`	Python has detected an error in its own code, or in an extension module. Please report this to the maintainers of your Python version, or of the extension in question, including the error message, the exact Python version (`sys.version`), and, if possible, your program's source code.
`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. Subclasses `NameError`.

Exception class	Raised when
`UnicodeError`	An error occurred while converting Unicode (i.e., a `str`) to a byte string, or vice versa. Subclasses `ValueError`.
`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.
`ZeroDivisionError`	A divisor (the righthand operand of a `/`, `//`, or `%` operator, or the second argument to the built-in function `divmod`) is `0`. Subclasses `ArithmeticError`.

OSError subclasses

`OSError` represents errors detected by the operating system. To handle such errors more elegantly, `OSError` has many subclasses, whose instances are what actually get raised; for a complete list, see Python's **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, and propagate any other exception that makes the file unreadable (except for the file not existing). Using an existing `OSError` subclass, you can accomplish the task quite simply:

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

The `FileNotFoundError` subclass of `OSError` makes this kind of common task simple and direct to express in code.

Exceptions “wrapping” other exceptions or tracebacks

Sometimes, you cause an exception while trying to handle another. To let you clearly diagnose this issue, each exception instance holds its own traceback object; you can make another exception instance with a different traceback with the `with_traceback` method.

Moreover, Python automatically stores which exception it’s handling as the `__context__` attribute of any further exception raised during the handling (unless you set the exception’s `__suppress_context__` attribute to **`True`** with the **`raise...from`** statement, which we cover shortly). If the new exception propagates, Python’s error message uses that exception’s `__context__` attribute to show details of the problem. For example, take the (deliberately!) broken code:

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

The error displayed is:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

During 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'
```

Thus, Python clearly displays both exceptions, the original and the intervening one.

To get more control over the error display, you can, if you wish, use the **`raise...from`** statement. When you execute **`raise`** *e* **`from`** *ex*, both *e* and *ex* are exception objects: *e* is the one that propagates, and *ex* is its "cause." Python records *ex* as the value of `e.__cause__`, and sets `e.__suppress_context__` to true. (Alternatively, *ex* can be **`None`**: then, Python sets `e.__cause__` to **`None`**, but still sets `e.__suppress_context__` to true, and thus leaves `e.__context__` alone).

As another example, here's a class implementing a mock filesystem directory using a Python dict, with the filenames as the keys and the file contents as the values:

```
class FileSystemDirectory:
    def __init__(self):
        self._files = {}

    def write_file(self, filename, contents):
        self._files[filename] = contents

    def read_file(self, filename):
        try:
            return self._files[filename]
        except KeyError:
            raise FileNotFoundError(filename)
```

When `read_file` is called with a nonexistent filename, the access to the `self._files` `dict` raises `KeyError`. Since this code is intended to emulate a filesystem directory, `read_file` catches the `KeyError` and raises `FileNotFoundError` instead.

As is, accessing a nonexistent file named `'data.txt'` will output an exception message similar to:

```
Traceback (most recent call last):
  File "C:\dev\python\faux_fs.py", line 11, in read_file
    return self._files[filename]
KeyError: 'data.txt'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "C:\dev\python\faux_fs.py", line 20, in <module>
    print(fs.read_file("data.txt"))
  File "C:\dev\python\faux_fs.py", line 13, in read_file
    raise FileNotFoundError(filename)
FileNotFoundError: data.txt
```

This exception report shows both the `KeyError` and the `FileNotFoundError`. To suppress the internal `KeyError` exception (to hide implementation details of `FileSystemDirectory`), we change the **`raise`** statement in `read_file` to:

```
    raise FileNotFoundError(filename) from None
```

Now the exception only shows the `FileNotFoundError` information:

```
Traceback (most recent call last):
  File "C:\dev\python\faux_fs.py", line 20, in <module>
    print(fs.read_file("data.txt"))
  File "C:\dev\python\faux_fs.py", line 13, in read_file
    raise FileNotFoundError(filename) from None
FileNotFoundError: data.txt
```

For details and motivations regarding exception chaining and embedding, 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 InvalidAttributeError(AttributeError):
    """Used to indicate attributes that could never be valid."""
```

AN EMPTY CLASS OR FUNCTION SHOULD HAVE A DOCSTRING

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 if nothing else!) 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 usually to have a docstring and no **`pass`** statement.

Given the semantics of **`try`**/**`except`**, raising an instance of a custom exception class such as `InvalidAttributeError` is almost the same as raising an instance of its standard exception superclass, `AttributeError`, but with some advantages. Any **`except`** clause that can handle `AttributeError` can handle `InvalidAttributeError` just as well. In addition, client code that knows about your `InvalidAttributeError` 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, suppose you write code like the following:

```
class SomeFunkyClass:
    """much hypothetical functionality snipped"""
    def __getattr__(self, name):
        """only clarifies the kind of attribute error"""
        if name.startswith('_'):
            raise InvalidAttributeError(
                f'Unknown private attribute {name!r}'
```

```
            )
        else:
            raise AttributeError(f'Unknown attribute {name!r}')
```

Now, client code can, if it so chooses, be more selective in its handlers. For example:

```
s = SomeFunkyClass()
try:
    value = getattr(s, thename)
except InvalidAttributeError as err:
    warnings.warn(str(err), stacklevel=2)
    value = None
# other cases of AttributeError just propagate, as they're unexpected
```

USE CUSTOM EXCEPTION CLASSES

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

Custom Exceptions and Multiple Inheritance

An effective approach to the use of 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, an instance of `CustomAttributeError` can only be raised explicitly and deliberately, showing an error related specifically to your code that *also* happens to be an `AttributeError`. When your code raises an in-

stance 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, specifically, by your module.

USE MULTIPLE INHERITANCE FOR CUSTOM EXCEPTIONS

Whenever you must decide whether to raise an instance of a specific standard exception, such as `AttributeError`, or of a custom exception class you define in your module, consider this multiple inheritance approach, which, in this book's authors' opinion,[2] gives you the best of both worlds in such cases. Make sure you clearly document this aspect of your module, because the technique, although handy, 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"**. We cover the main cases of such exception classes throughout the rest of this book, in chapters covering the standard library modules that supply and may raise them.

ExceptionGroup and except*

3.11+ In some circumstances, such as when performing validation of some input data against multiple criteria, it is useful to be able to raise more than a single exception at once. Python 3.11 introduced a mechanism to raise multiple exceptions at once using an `ExceptionGroup` instance and to process more than one exception using an **`except*`** form in place of **`except`**.

To raise `ExceptionGroup`, the validating code captures multiple `Exceptions` into a list and then raises an `ExceptionGroup` that is constructed using that list. Here is some code that searches for misspelled and invalid words, and raises an `ExceptionGroup` containing all of the found errors:

```
class GrammarError(Exception):
    """Base exception for grammar checking"""
    def __init__(self, found, suggestion):
        self.found = found
        self.suggestion = suggestion

class InvalidWordError(GrammarError):
    """Misused or nonexistent word"""

class MisspelledWordError(GrammarError):
    """Spelling error"""

invalid_words = {
    'irregardless': 'regardless',
    "ain't": "isn't",
}
misspelled_words = {
    'tacco': 'taco',
}

def check_grammar(s):
    exceptions = []
    for word in s.lower().split():
        if (suggestion := invalid_words.get(word)) is not None:
            exceptions.append(InvalidWordError(word, suggestion))
        elif (suggestion := misspelled_words.get(word)) is not None:
            exceptions.append(MisspelledWordError(word, suggestion))
    if exceptions:
        raise ExceptionGroup('Found grammar errors', exceptions)
```

The following code validates a sample text string and lists out all the found errors:

```
text = "Irregardless a hot dog ain't a tacco"
try:
    check_grammar(text)
except* InvalidWordError as iwe:
    print('\n'.join(f'{e.found!r} is not a word, use {e.suggestion!r}'
                    for e in iwe.exceptions))
except* MisspelledWordError as mwe:
    print('\n'.join(f'Found {e.found!r}, perhaps you meant'
                    f' {e.suggestion!r}?'
                    for e in mwe.exceptions))
else:
    print('No errors!')
```

giving this output:

```
'irregardless' is not a word, use 'regardless'
"ain't" is not a word, use "isn't"
Found 'tacco', perhaps you meant 'taco'?
```

Unlike **except**, after it finds an initial match, **except*** continues to look for additional exception handlers matching exception types in the raised `ExceptionGroup`.

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 OK.
- The work needed for checking purposes may duplicate a substantial part of the work done in the operation itself.
- The programmer might easily err by omitting a needed check.
- The situation might change between the moment when you perform the checks and the moment when, later (even by a tiny fraction of a second!), you attempt the operation.

The preferred idiom in Python is to attempt the operation in a **`try`** clause and handle the exceptions that may result in one or more **`except`** clauses. This idiom is known as **"It's easier to ask forgiveness than permission" (EAFP)**, a frequently quoted 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 using the LBYL idiom:

```
def safe_divide_1(x, y):
    if y==0:
        print('Divide-by-0 attempt 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 using the EAFP idiom:

```
def safe_divide_2(x, y):
    try:
        return x/y
    except ZeroDivisionError:
```

```
        print('Divide-by-0 attempt detected')
        return None
```

With EAFP, the mainstream case is up front in a **try** clause, and the anomalies are handled in the following **except** clause, making the whole function easier to read and understand.

EAFP is a good error-handling strategy, but it is not a panacea. In particular, you must take care not to 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 8-2**):

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

The intention of the `trycalling` function is to try calling a method named *`attrib`* on the 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 accidentally hides any error case where an `AttributeError` is raised inside the sought-after method, silently returning *`default`* in those cases. This could 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 exception. The proper approach to EAFP involves frequent use of the `else` clause in `try/except` statements (which is more explicit, and thus better Python 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 broad, particularly once you have convinced yourself of the power of EAFP as a general error-checking strategy. A `try/except` combination is too broad 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 may 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 “edge” 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, 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. Your **`try/except`** clauses should still be reasonably narrow, although this is not quite as crucial as when you use EAFP to structure your handling of not-really-erroneous special/edge 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 dire 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 to yourself 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**; 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 will help you diagnose and debug the problem and direct that information to whatever destinations you please. For example, you might use the `traceback` module (covered in **"The traceback Module"**) to format stack traces. When your exception-reporting function terminates, so does your program.

The logging module

The Python standard library offers the rich and powerful `logging` module to let you organize the logging of messages from your applications in systematic, flexible ways. Pushing things to the limit, you might write a whole hierarchy of `Logger` classes and subclasses; you could couple the loggers with instances of `Handler` (and subclasses thereof), or insert instances of the class `Filter` to fine-tune criteria determining what messages get logged in which ways.

Messages are formatted by instances of the `Formatter` class—the messages themselves are instances of the `LogRecord` class. The `logging` module 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` module sports a frighteningly complex and powerful architecture, suitable for implementing highly sophisticated logging strategies and policies that may be needed in vast and complicated software systems, in most applications you might get away with using a tiny subset of the package. First, **`import`** `logging`. Then, emit your message by passing it as a string to any of the module's 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 logging level). If `foo` is used only for logging and is especially compute- or I/O-intensive to create, you can use `isEnabledFor` to conditionalize the expensive code that creates `foo`:

```
if logging.getLogger().isEnabledFor(logging.DEBUG):
    foo = cpu_intensive_function()
    logging.debug('foo is %r', foo)
```

Configuring logging

Unfortunately, the `logging` module does not support the more readable formatting approaches covered in **"String Formatting"**, but only the legacy one mentioned in the previous subsection. Fortunately, it's very rare to need any formatting specifiers beyond `%s` (which calls `__str__`) and `%r` (which calls `__repr__`).

By default, the threshold level is `WARNING`: any of the functions `warning`, `error`, or `critical` results in logging output, but the functions `debug` and `info` do not. To change the threshold level at any time, call `logging.getLogger().setLevel`, passing as the only argument one of the

corresponding constants supplied by the `logging` module: `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 the 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 until you change the level again, and so forth.

By default, logging output goes to your process's standard error stream (`sys.stderr`, as covered in **Table 8-3**) 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 things up 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 writes logging messages to a file, nicely formatted with a precise human-readable timestamp, followed by the severity level right-aligned in an eight-character field, followed by the message proper.

For much, much more detailed information on the `logging` module and all the wonders you can perform with it, be sure to consult Python's **rich online documentation**.

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.[3]

`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.

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 done using **`assert`**, since it can be turned into a null operation by a Python command-line flag.

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).

THE __DEBUG__ BUILT-IN VARIABLE

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

---

1 Except that multiple calls to `close` are allowed and innocuous: all but the first one perform no operation.

2 This is somewhat controversial: while this book's authors agree on this being "best practice," some others strongly insist that one should always avoid multiple inheritance, including in this specific case.

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