CHAPTER 15
Context Managers and else Blocks

> Context managers may end up being almost as important as the subroutine itself. We've only scratched the surface with them. […] Basic has a `with` statement, there are `with` statements in lots of languages. But they don't do the same thing, they all do something very shallow, they save you from repeated dotted [attribute] lookups, they don't do setup and tear down. Just because it's the same name don't think it's the same thing. The `with` statement is a very big deal.[1]
>
> — Raymond Hettinger
> *Eloquent Python evangelist*

In this chapter, we will discuss control flow features that are not so common in other languages, and for this reason tend to be overlooked or underused in Python. They are:

- The `with` statement and context managers
- The `else` clause in `for`, `while`, and `try` statements

The `with` statement sets up a temporary context and reliably tears it down, under the control of a context manager object. This prevents errors and reduces boilerplate code, making APIs at the same time safer and easier to use. Python programmers are finding lots of uses for `with` blocks beyond automatic file closing.

The `else` clause is completely unrelated to `with`. But this is Part V, and I couldn't find another place for covering `else`, and I wouldn't have a one-page chapter about it, so here it is.

Let's review the smaller topic to get to the real substance of this chapter.

1. PyCon US 2013 keynote: "What Makes Python Awesome"; the part about `with` starts at 23:00 and ends at 26:15.

WOW! eBook
www.wowebook.org

Do This, Then That: else Blocks Beyond if

This is no secret, but it is an underappreciated language feature: the `else` clause can be used not only in `if` statements but also in `for`, `while`, and `try` statements.

The semantics of `for/else`, `while/else`, and `try/else` are closely related, but very different from `if/else`. Initially the word `else` actually hindered my understanding of these features, but eventually I got used to it.

Here are the rules:

`for`
: The `else` block will run only if and when the `for` loop runs to completion (i.e., not if the `for` is aborted with a `break`).

`while`
: The `else` block will run only if and when the `while` loop exits because the condition became *falsy* (i.e., not when the `while` is aborted with a `break`).

`try`
: The `else` block will only run if no exception is raised in the `try` block. The official docs also state: "Exceptions in the `else` clause are not handled by the preceding `except` clauses."

In all cases, the `else` clause is also skipped if an exception or a `return`, `break`, or `continue` statement causes control to jump out of the main block of the compound statement.

I think `else` is a very poor choice for the keyword in all cases except `if`. It implies an excluding alternative, like "Run this loop, otherwise do that," but the semantics for `else` in loops is the opposite: "Run this loop, then do that." This suggests `then` as a better keyword—which would also make sense in the `try` context: "Try this, then do that." However, adding a new keyword is a breaking change to the language, and Guido avoids it like the plague.

Using `else` with these statements often makes the code easier to read and saves the trouble of setting up control flags or adding extra `if` statements.

The use of `else` in loops generally follows the pattern of this snippet:

```
for item in my_list:
    if item.flavor == 'banana':
        break
else:
    raise ValueError('No banana flavor found!')
```


WOW! eBook
www.wowebook.org

In the case of `try/except` blocks, `else` may seem redundant at first. After all, the `after_call()` in the following snippet will run only if the `dangerous_call()` does not raise an exception, correct?

```
try:
    dangerous_call()
    after_call()
except OSError:
    log('OSError...')
```

However, doing so puts the `after_call()` inside the `try` block for no good reason. For clarity and correctness, the body of a `try` block should only have the statements that may generate the expected exceptions. This is much better:

```
try:
    dangerous_call()
except OSError:
    log('OSError...')
else:
    after_call()
```

Now it's clear that the `try` block is guarding against possible errors in `dangerous_call()` and not in `after_call()`. It's also more obvious that `after_call()` will only execute if no exceptions are raised in the `try` block.

In Python, `try/except` is commonly used for control flow, and not just for error handling. There's even an acronym/slogan for that documented in the official Python glossary:

> *EAFP*
>
> Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

The glossary then defines LBYL:

> *LBYL*
>
> Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many if statements. In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code, if key in mapping: return mapping[key] can fail if another thread removes key from mapping after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

Given the EAFP style, it makes even more sense to know and use well `else` blocks in `try/except` statements.

Now let's address the main topic of this chapter: the powerful `with` statement.

WOW! eBook
www.wowebook.org

Context Managers and with Blocks

Context manager objects exist to control a `with` statement, just like iterators exist to control a `for` statement.

The `with` statement was designed to simplify the `try/finally` pattern, which guarantees that some operation is performed after a block of code, even if the block is aborted because of an exception, a `return` or `sys.exit()` call. The code in the `finally` clause usually releases a critical resource or restores some previous state that was temporarily changed.

The context manager protocol consists of the `__enter__` and `__exit__` methods. At the start of the `with`, `__enter__` is invoked on the context manager object. The role of the `finally` clause is played by a call to `__exit__` on the context manager object at the end of the `with` block.

The most common example is making sure a file object is closed. See Example 15-1 for a detailed demonstration of using `with` to close a file.

Example 15-1. Demonstration of a file object as a context manager

```
>>> with open('mirror.py') as fp:  # ❶
...     src = fp.read(60)  # ❷
...
>>> len(src)
60
>>> fp  # ❸
<_io.TextIOWrapper name='mirror.py' mode='r' encoding='UTF-8'>
>>> fp.closed, fp.encoding  # ❹
(True, 'UTF-8')
>>> fp.read(60)  # ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

❶ `fp` is bound to the opened file because the file's `__enter__` method returns `self`.

❷ Read some data from `fp`.

❸ The `fp` variable is still available.[2]

❹ You can read the attributes of the `fp` object.

❺ But you can't perform I/O with `fp` because at the end of the `with` block, the `TextIOWrapper.__exit__` method is called and closes the file.

2. `with` blocks don't define a new scope, as functions and modules do.

WOW! eBook
www.wowebook.org

The first callout in Example 15-1 makes a subtle but crucial point: the context manager object is the result of evaluating the expression after `with`, but the value bound to the target variable (in the `as` clause) is the result of calling `__enter__` on the context manager object.

It just happens that in Example 15-1, the `open()` function returns an instance of `TextIOWrapper`, and its `__enter__` method returns `self`. But the `__enter__` method may also return some other object instead of the context manager.

When control flow exits the `with` block in any way, the `__exit__` method is invoked on the context manager object, not on whatever is returned by `__enter__`.

The `as` clause of the `with` statement is optional. In the case of `open`, you'll always need it to get a reference to the file, but some context managers return `None` because they have no useful object to give back to the user.

Example 15-2 shows the operation of a perfectly frivolous context manager designed to highlight the distinction between the context manager and the object returned by its `__enter__` method.

Example 15-2. Test driving the LookingGlass context manager class

```
>>> from mirror import LookingGlass
>>> with LookingGlass() as what:   ❶
...     print('Alice, Kitty and Snowdrop')   ❷
...     print(what)
...
pordwonS dna yttiK ,ecilA   ❸
YKCOWREBBAJ
>>> what   ❹
'JABBERWOCKY'
>>> print('Back to normal.')   ❺
Back to normal.
```

❶ The context manager is an instance of `LookingGlass`; Python calls `__enter__` on the context manager and the result is bound to `what`.

❷ Print a `str`, then the value of the target variable `what`.

❸ The output of each `print` comes out backward.

❹ Now the `with` block is over. We can see that the value returned by `__enter__`, held in `what`, is the string `'JABBERWOCKY'`.

❺ Program output is no longer backward.

Example 15-3 shows the implementation of `LookingGlass`.

WOW! eBook
www.wowebook.org

Example 15-3. mirror.py: code for the LookingGlass context manager class

```
class LookingGlass:

    def __enter__(self):   ❶
        import sys
        self.original_write = sys.stdout.write   ❷
        sys.stdout.write = self.reverse_write   ❸
        return 'JABBERWOCKY'   ❹

    def reverse_write(self, text):   ❺
        self.original_write(text[::-1])

    def __exit__(self, exc_type, exc_value, traceback):   ❻
        import sys   ❼
        sys.stdout.write = self.original_write   ❽
        if exc_type is ZeroDivisionError:   ❾
            print('Please DO NOT divide by zero!')
            return True   ❿
        ⓫
```

❶ Python invokes `__enter__` with no arguments besides `self`.

❷ Hold the original `sys.stdout.write` method in an instance attribute for later use.

❸ Monkey-patch `sys.stdout.write`, replacing it with our own method.

❹ Return the `'JABBERWOCKY'` string just so we have something to put in the target variable `what`.

❺ Our replacement to `sys.stdout.write` reverses the `text` argument and calls the original implementation.

❻ Python calls `__exit__` with `None, None, None` if all went well; if an exception is raised, the three arguments get the exception data, as described next.

❼ It's cheap to import modules again because Python caches them.

❽ Restore the original method to `sys.stdout.write`.

❾ If the exception is not `None` and its type is `ZeroDivisionError`, print a message…

❿ …and return `True` to tell the interpreter that the exception was handled.

⓫ If `__exit__` returns `None` or anything but `True`, any exception raised in the `with` block will be propagated.

When real applications take over standard output, they often want to replace `sys.stdout` with another file-like object for a while, then switch back to the original. The `contextlib.redirect_stdout` context manager does exactly that: just pass it the file-like object that will stand in for `sys.stdout`.

WOW! eBook
www.wowebook.org

The interpreter calls the `__enter__` method with no arguments—beyond the implicit `self`. The three arguments passed to `__exit__` are:

`exc_type`
: The exception class (e.g., `ZeroDivisionError`).

`exc_value`
: The exception instance. Sometimes, parameters passed to the exception constructor—such as the error message—can be found in `exc_value.args`.

`traceback`
: A `traceback` object.[3]

For a detailed look at how a context manager works, see Example 15-4, where `LookingGlass` is used outside of a `with` block, so we can manually call its `__enter__` and `__exit__` methods.

Example 15-4. Exercising LookingGlass without a with block

```
>>> from mirror import LookingGlass
>>> manager = LookingGlass()   ❶
>>> manager
<mirror.LookingGlass object at 0x2a578ac>
>>> monster = manager.__enter__()   ❷
>>> monster == 'JABBERWOCKY'   ❸
eurT
>>> monster
'YKCOWREBBAJ'
>>> manager
>ca875a2x0 ta tcejbo ssalGgnikooL.rorrim<
>>> manager.__exit__(None, None, None)   ❹
>>> monster
'JABBERWOCKY'
```

❶ Instantiate and inspect the `manager` instance.

❷ Call the context manager `__enter__()` method and store result in `monster`.

❸ Monster is the string `'JABBERWOCKY'`. The `True` identifier appears reversed because all output via `stdout` goes through the `write` method we patched in `__enter__`.

❹ Call `manager.__exit__` to restore previous `stdout.write`.

3. The three arguments received by `self` are exactly what you get if you call `sys.exc_info()` in the `finally` block of a `try/finally` statement. This makes sense, considering that the `with` statement is meant to replace most uses of `try/finally`, and calling `sys.exc_info()` was often necessary to determine what clean-up action would be required.

WOW! eBook
www.wowebook.org

Context managers are a fairly novel feature and slowly but surely the Python community is finding new, creative uses for them. Some examples from the standard library are:

- Managing transactions in the `sqlite3` module; see "12.6.7.3. Using the connection as a context manager".
- Holding locks, conditions, and semaphores in `threading` code; see "17.1.10. Using locks, conditions, and semaphores in the `with` statement".
- Setting up environments for arithmetic operations with `Decimal` objects; see the `decimal.localcontext` documentation.
- Applying temporary patches to objects for testing; see the `unittest.mock.patch` function.

The standard library also includes the `contextlib` utilities, covered next.

The contextlib Utilities

Before rolling your own context manager classes, take a look at "29.6 `contextlib` — Utilities for `with`-statement contexts" in *The Python Standard Library*. Besides the already mentioned `redirect_stdout`, the `contextlib` module includes classes and other functions that are more widely applicable:

`closing`
: A function to build context managers out of objects that provide a `close()` method but don't implement the `__enter__`/`__exit__` protocol.

`suppress`
: A context manager to temporarily ignore specified exceptions.

`@contextmanager`
: A decorator that lets you build a context manager from a simple generator function, instead of creating a class and implementing the protocol.

`ContextDecorator`
: A base class for defining class-based context managers that can also be used as function decorators, running the entire function within a managed context.

`ExitStack`
: A context manager that lets you enter a variable number of context managers. When the `with` block ends, `ExitStack` calls the stacked context managers' `__exit__` methods in LIFO order (last entered, first exited). Use this class when you don't know beforehand how many context managers you need to enter in your `with` block; for example, when opening all files from an arbitrary list of files at the same time.

WOW! eBook
www.wowebook.org

The most widely used of these utilities is surely the `@contextmanager` decorator, so it deserves more attention. That decorator is also intriguing because it shows a use for the `yield` statement unrelated to iteration. This paves the way to the concept of a coroutine, the theme of the next chapter.

Using @contextmanager

The `@contextmanager` decorator reduces the boilerplate of creating a context manager: instead of writing a whole class with `__enter__`/`__exit__` methods, you just implement a generator with a single `yield` that should produce whatever you want the `__enter__` method to return.

In a generator decorated with `@contextmanager`, `yield` is used to split the body of the function in two parts: everything before the `yield` will be executed at the beginning of the `while` block when the interpreter calls `__enter__`; the code after `yield` will run when `__exit__` is called at the end of the block.

Here is an example. Example 15-5 replaces the `LookingGlass` class from Example 15-3 with a generator function.

Example 15-5. mirror_gen.py: a context manager implemented with a generator

```
import contextlib


@contextlib.contextmanager    ❶
def looking_glass():
    import sys
    original_write = sys.stdout.write    ❷

    def reverse_write(text):    ❸
        original_write(text[::-1])

    sys.stdout.write = reverse_write    ❹
    yield 'JABBERWOCKY'    ❺
    sys.stdout.write = original_write    ❻
```

❶ Apply the `contextmanager` decorator.

❷ Preserve original `sys.stdout.write` method.

❸ Define custom `reverse_write` function; `original_write` will be available in the closure.

❹ Replace `sys.stdout.write` with `reverse_write`.

❺ Yield the value that will be bound to the target variable in the `as` clause of the `with` statement. This function pauses at this point while the body of the `with` executes.

WOW! eBook
www.wowebook.org

❻ When control exits the `with` block in any way, execution continues after the `yield`; here the original `sys.stdout.write` is restored.

Example 15-6 shows the `looking_glass` function in operation.

Example 15-6. Test driving the looking_glass context manager function

```
>>> from mirror_gen import looking_glass
>>> with looking_glass() as what:   ❶
...     print('Alice, Kitty and Snowdrop')
...     print(what)
...
pordwonS dna yttiK ,ecilA
YKCOWREBBAJ
>>> what
'JABBERWOCKY'
```

❶ The only difference from Example 15-2 is the name of the context manager: `looking_glass` instead of `LookingGlass`.

Essentially the `contextlib.contextmanager` decorator wraps the function in a class that implements the `__enter__` and `__exit__` methods.[4]

The `__enter__` method of that class:

1. Invokes the generator function and holds on to the generator object—let's call it `gen`.
2. Calls `next(gen)` to make it run to the `yield` keyword.
3. Returns the value yielded by `next(gen)`, so it can be bound to a target variable in the with/as form.

When the `with` block terminates, the `__exit__` method:

1. Checks an exception was passed as `exc_type`; if so, `gen.throw(exception)` is invoked, causing the exception to be raised in the `yield` line inside the generator function body.
2. Otherwise, `next(gen)` is called, resuming the execution of the generator function body after the `yield`.

Example 15-5 has a serious flaw: if an exception is raised in the body of the `with` block, the Python interpreter will catch it and raise it again in the `yield` expression inside `looking_glass`. But there is no error handling there, so the `looking_glass` function

4. The actual class is named `_GeneratorContextManager`. If you want to see exactly how it works, read its source code in *Lib/contextlib.py* in the Python 3.4 distribution.

WOW! eBook
www.wowebook.org

will abort without ever restoring the original `sys.stdout.write` method, leaving the system in an invalid state.

Example 15-7 adds special handling of the `ZeroDivisionError` exception, making it functionally equivalent to the class-based Example 15-3.

Example 15-7. mirror_gen_exc.py: generator-based context manager implementing exception handling—same external behavior as Example 15-3

```
import contextlib


@contextlib.contextmanager
def looking_glass():
    import sys
    original_write = sys.stdout.write

    def reverse_write(text):
        original_write(text[::-1])

    sys.stdout.write = reverse_write
    msg = ''    ❶
    try:
        yield 'JABBERWOCKY'
    except ZeroDivisionError:    ❷
        msg = 'Please DO NOT divide by zero!'
    finally:
        sys.stdout.write = original_write    ❸
        if msg:
            print(msg)    ❹
```

❶ Create a variable for a possible error message; this is the first change in relation to Example 15-5.

❷ Handle `ZeroDivisionError` by setting an error message.

❸ Undo monkey-patching of `sys.stdout.write`.

❹ Display error message, if it was set.

Recall that the `__exit__` method tells the interpreter that it has handled the exception by returning `True`; in that case, the interpreter suppresses the exception. On the other hand, if `__exit__` does not explicitly return a value, the interpreter gets the usual `None`, and propagates the exception. With `@contextmanager`, the default behavior is inverted: the `__exit__` method provided by the decorator assumes any exception sent into the generator is handled and should be suppressed.[5] You must explicitly re-raise an

5. The exception is sent into the generator using the `throw` method, covered in "Coroutine Termination and Exception Handling" on page 471.

WOW! eBook
www.wowebook.org

exception in the decorated function if you don't want `@contextmanager` to suppress it.[6]

Having a `try/finally` (or a `with` block) around the `yield` is an unavoidable price of using `@contextmanager`, because you never know what the users of your context manager are going to do inside their `with` block.[7]

An interesting real-life example of `@contextmanager` outside of the standard library is Martijn Pieters' in-place file rewriting context manager. Example 15-8 shows how it's used.

Example 15-8. A context manager for rewriting files in place

```
import csv

with inplace(csvfilename, 'r', newline='') as (infh, outfh):
    reader = csv.reader(infh)
    writer = csv.writer(outfh)

    for row in reader:
        row += ['new', 'columns']
        writer.writerow(row)
```

The `inplace` function is a context manager that gives you two handles—`infh` and `outfh` in the example—to the same file, allowing your code to read and write to it at the same time. It's easier to use than the standard library's `fileinput.input` function (which also provides a context manager, by the way).

If you want to study Martijn's `inplace` source code (listed in the post), find the `yield` keyword: everything before it deals with setting up the context, which entails creating a backup file, then opening and yielding references to the readable and writable file handles that will be returned by the `__enter__` call. The `__exit__` processing after the `yield` closes the file handles and restores the file from the backup if something went wrong.

Note that the use of `yield` in a generator used with the `@contextmanager` decorator has nothing to do with iteration. In the examples shown in this section, the generator function is operating more like a coroutine: a procedure that runs up to a point, then sus-

6. This convention was adopted because when context managers were created, generators could not `return` values, only `yield`. They now can, as explained in "Returning a Value from a Coroutine" on page 475. As you'll see, returning a value from a generator does involve an exception.

7. This tip is quoted literally from a comment by Leonardo Rochael, one of the tech reviewers for this book. Nicely said, Leo!

WOW! eBook
www.wowebook.org

pends to let the client code run until the client wants the coroutine to proceed with its job. Chapter 16 is all about coroutines.

Chapter Summary

This chapter started easily enough with discussion of `else` blocks in `for`, `while`, and `try` statements. Once you get used to the peculiar meaning of the `else` clause in these statements, I believe `else` can clarify your intentions.

We then covered context managers and the meaning of the `with` statement, quickly moving beyond its common use to automatically close opened files. We implemented a custom context manager: the `LookingGlass` class with the `__enter__`/`__exit__` methods, and saw how to handle exceptions in the `__exit__` method. A key point that Raymond Hettinger made in his PyCon US 2013 keynote is that `with` is not just for resource management, but it's a tool for factoring out common setup and teardown code, or any pair of operations that need to be done before and after another procedure (slide 21, What Makes Python Awesome?).

Finally, we reviewed functions in the `contextlib` standard library module. One of them, the `@contextmanager` decorator, makes it possible to implement a context manager using a simple generator with one `yield`—a leaner solution than coding a class with at least two methods. We reimplemented the `LookingGlass` as a `looking_glass` generator function, and discussed how to do exception handling when using `@contextmanager`.

The `@contextmanager` decorator is an elegant and practical tool that brings together three distinctive Python features: a function decorator, a generator, and the `with` statement.

Further Reading

Chapter 8, "Compound Statements," in *The Python Language Reference* says pretty much everything there is to say about `else` clauses in `if`, `for`, `while`, and `try` statements. Regarding Pythonic usage of `try/except`, with or without `else`, Raymond Hettinger has a brilliant answer to the question "Is it a good practice to use try-except-else in Python?" in StackOverflow. Alex Martelli's *Python in a Nutshell, 2E* (O'Reilly), has a chapter about exceptions with an excellent discussion of the EAFP style, crediting computing pioneer Grace Hopper for coining the phrase "It's easier to ask forgiveness than permission."

The *Python Standard Library*, Chapter 4, "Built-in Types," has a section devoted to Context Manager Types. The `__enter__`/`__exit__` special methods are also documented in *The Python Language Reference* in "3.3.8. With Statement Context Managers". Context managers were introduced in PEP 343 — The "with" Statement. This PEP

WOW! eBook
www.wowebook.org

is not easy reading because it spends a lot of time covering corner cases and arguing against alternative proposals. That's the nature of PEPs.

Raymond Hettinger highlighted the `with` statement as a "winning language feature" in his PyCon US 2013 keynote. He also showed some interesting applications of context managers in his talk "Transforming Code into Beautiful, Idiomatic Python" at the same conference.

Jeff Preshing' blog post "The Python `with` Statement by Example" is interesting for the examples using context managers with the `pycairo` graphics library.

Beazley and Jones devised context managers for very different purposes in their *Python Cookbook, 3E* (O'Reilly). "Recipe 8.3. Making Objects Support the Context-Management Protocol" implements a `LazyConnection` class whose instances are context managers that open and close network connections automatically in `with` blocks. "Recipe 9.22. Defining Context Managers the Easy Way" introduces a context manager for timing code, and another for making transactional changes to a `list` object: within the `with` block, a working copy of the `list` instance is made, and all changes are applied to that working copy. Only when the `with` block completes without an exception, the working copy replaces the original list. Simple and ingenious.

Soapbox

Factoring Out the Bread

In his PyCon US 2013 keynote, "What Makes Python Awesome," Raymond Hettinger says when he first saw the `with` statement proposal he thought it was "a little bit arcane." Initially, I had a similar reaction. PEPs are often hard to read, and PEP 343 is typical in that regard.

Then—Hettinger told us—he had an insight: subroutines are the most important invention in the history of computer languages. If you have sequences of operations like A;B;C and P;B;Q, you can factor out B in a subroutine. It's like factoring out the filling in a sandwich: using tuna with different breads. But what if you want to factor out the bread, to make sandwiches with wheat bread, using a different filling each time? That's what the `with` statement offers. It's the complement of the subroutine. Hettinger went on to say:

> The `with` statement is a very big deal. I encourage you to go out and take this tip of the iceberg and drill deeper. You can probably do profound things with the `with` statement. The best uses of it have not been discovered yet. I expect that if you make good use of it, it will be copied into other languages and all future languages will have it. You can be part of discovering something almost as profound as the invention of the subroutine itself.

WOW! eBook
www.wowebook.org

Hettinger admits he is overselling the `with` statement. Nevertheless, it is a very useful feature. When he used the sandwich analogy to explain how `with` is the complement to the subroutine, many possibilities opened up in my mind.

If you need to convince anyone that Python is awesome, you should watch Hettinger's keynote. The bit about context managers is from 23:00 to 26:15. But the entire keynote is excellent.

WOW! eBook
www.wowebook.org