08. Context Managers

October 14, 2025

0.1 Context Managers - `with` statement

Context managers in Python provide a way to ensure that resources are properly managed, especially when dealing with operations like file I/O, network connections, or locking and unlocking mechanisms. They are most commonly used with the `with` statement, which ensures that the setup and teardown of resources are handled automatically.

0.1.1 Basic Example with a File

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent `try-finally` blocks:

```
with open('workfile') as f:
    read_data = f.read()
```

The equivalent `try-finally` block of the `with` block above would be:

```
f = open('workfile')
try:
    read_data = f.read()
finally:
    f.close()
```

While comparing it to the first example we can see that a lot of boilerplate code is eliminated just by using `with`. The main advantage of using a with statement is that it makes sure our file is closed without paying attention to how the nested block exits. Another advantage is that the file is closed even if an exception is raised while processing it.

The `with` statement is used to wrap the execution of a block with methods defined by a context manager. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. This is done through the `__enter__` and `__exit__` dunder methods.

For file objects: - `__enter__` returns the file object that is captured in the variable following `as` - `__exit__` closes the file

[8]:
```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

```
hello world
```

[9]:
```
file.closed  # check if file object is closed after `with` block
```

[9]:
```
True
```

0.1.2 Why Use Context Managers?

- **Resource Management**: Ensures resources like files or network connections are properly released after use.
- **Error Handling**: Handles exceptions and ensures the resource is still cleaned up properly.
- **Code Clarity**: Makes the code more readable and concise by abstracting setup and teardown logic.

0.1.3 Custom Context Managers

You can also create your own context managers using the `contextlib` module or by defining a class with `__enter__` and `__exit__` methods.

Example with `contextlib`:

[10]:
```
from contextlib import contextmanager

@contextmanager
def managed_resource():
    print("Setup resource")
    try:
        yield "resource"
    finally:
        print("Teardown resource")
```

[11]:
```
with managed_resource() as r:
    print("Using resource")
    print(r)
```

```
Setup resource
Using resource
resource
Teardown resource
```

Explanation:

1. `@contextmanager`:
   - This is a decorator that simplifies the creation of context managers.
2. `managed_resource` **function**:
   - The function is defined to manage a resource.
   - The code before the `yield` statement is the setup code.
   - The code after the `yield` statement is the teardown code.
3. **Using the custom context manager**:
   - The `with managed_resource()` statement calls the context manager.
   - It prints "Setup resource" before the `yield` and "Teardown resource" after the block inside the `with` statement is executed.

0.1.4 Example with Class-based Context Manager

[12]:
```
class ManagedResource:
    def __enter__(self):
        print("Setup resource")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Teardown resource")

with ManagedResource() as resource:
    print("Using resource")
    print(resource)
```

```
Setup resource
Using resource
<__main__.ManagedResource object at 0x106ae5160>
Teardown resource
```

Explanation:

1. **`__enter__` method**:
   - This method is called when the execution enters the context of the `with` statement.
   - It sets up the resource and returns it.
2. **`__exit__` method**:
   - This method is called when the execution leaves the context of the `with` statement.
   - It handles any necessary cleanup of the resource.
3. **Using the class-based context manager**:
   - The `with ManagedResource() as resource` statement calls the context manager.
   - It prints "Setup resource" when entering the block and "Teardown resource" when exiting the block, ensuring the resource is managed properly.

0.1.5 A more practical example

Let's say we want to redirect `stdout` to a file with a context manager:

[13]:
```
import sys
from contextlib import contextmanager


@contextmanager
def redirect_stdout(file):
    saved_out = sys.stdout
    sys.stdout = file
    yield
    sys.stdout = saved_out
```

This context manager takes a file object as parameter. In the setup code, you reassign the standard output, `sys.stdout`, to an instance attribute to avoid losing the reference to it. Then you reassign

the standard output to point to the file on your disk. In the teardown code, you just restore the standard output to its original value.

[14]:
```
with open("out.txt", "w") as f:
    with redirect_stdout(f):
        print("This is going to the file.")
        print("This is also going to out.txt.")
    print("Back to printing to stdout.")
```

```
Back to printing to stdout.
```

The same example, this time with class-based context manager:

[15]:
```
class RedirectStdout:
    def __init__(self, file):
        self.file = file

    def __enter__(self):
        self.saved_out = sys.stdout
        sys.stdout = self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.stdout = self.saved_out
```

[16]:
```
with open("out2.txt", "w") as f:
    with RedirectStdout(f):
        print("This is going to the file.")
    print("Back to printing to stdout.")
```

```
Back to printing to stdout.
```

0.1.6 Exercises

1. Create a context manager to measure the time taken by a block of code. Do it with both methods presented here (function and class based context managers).
2. The following code creates a database connection, creates a table and inserts some rows: “‘python import sqlite3

   con = sqlite3.connect(“tutorial.db”) cur = con.cursor() cur.execute(“CREATE TABLE movie(title, year, score)”) cur.execute(“ “ ” INSERT INTO movie VALUES (‘Monty Python and the Holy Grail’, 1975, 8.2), (‘And Now for Something Completely Different’, 1971, 7.5) “ “ “) con.commit() res = cur.execute(”SELECT score FROM movie”) print(res.fetchall())

   con.close() “‘

   Create a context manager to handle a database connection. The context manager will return the connection object on setup and will close the connection on teardown.