Python Context Manager Cheat Sheet

What are Context Managers?

Context managers are objects that define the runtime context for executing a block of code. They're commonly used with the `with` statement to ensure proper resource management, cleanup, and exception handling.

Core Methods

- `__enter__()`: Called when entering the `with` block
- `__exit__(exc_type, exc_value, traceback)`: Called when exiting the `with` block

Basic Usage

File Handling (Most Common Example)

```
with open('file.txt', 'r') as f:
    content = f.read()
# File is automatically closed here, even if an exception occurs
```

Creating Custom Context Managers

Method 1: Class-Based Approach

```
class DatabaseConnection:
    def __enter__(self):
        self.connection = connect_to_database()
        print("Database connection opened")
        return self.connection

    def __exit__(self, exc_type, exc_value, traceback):
        self.connection.close()
        print("Database connection closed")
        return False  # Don't suppress exceptions

# Usage
with DatabaseConnection() as db:
    db.execute("SELECT * FROM users")
```

Method 2: Using @contextmanager Decorator

```
from contextlib import contextmanager
import time
```

```python
@contextmanager
def timer():
    start = time.time()
    try:
        yield
    finally:
        end = time.time()
        print(f"Execution took {end - start:.2f} seconds")

# Usage
with timer():
    time.sleep(1)
    print("Some operation")
```

Understanding exit Parameters

```python
def __exit__(self, exc_type, exc_value, traceback):
    # exc_type: The exception type (None if no exception)
    # exc_value: The exception instance (None if no exception)
    # traceback: The traceback object (None if no exception)

    # Return True to suppress the exception
    # Return False (or None) to propagate it
    pass
```

Common Interview Questions & Answers

Q1: Multiple Context Managers in One Statement

Question: Can you use multiple context managers in one `with` statement?

Answer: Yes, you can chain them:

```python
with open('input.txt') as infile, open('output.txt', 'w') as outfile:
    outfile.write(infile.read())
```

Q2: Exception Handling

Question: How do you handle exceptions in context managers?

Answer:

```python
class ExceptionHandler:
    def __enter__(self):
        return self
```

```
    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type is ValueError:
            print(f"Handled ValueError: {exc_value}")
            return True  # Suppress the exception
        return False  # Let other exceptions propagate

# Usage
with ExceptionHandler():
    raise ValueError("This will be handled")
```

Q3: contextlib.closing()

Question: What's the difference between `contextlib.closing()` and regular context managers?

Answer: `contextlib.closing()` wraps objects that have a `close()` method but aren't context managers:

```
from contextlib import closing
import urllib.request

with closing(urllib.request.urlopen('http://example.com')) as page:
    data = page.read()
```

Q4: Reentrant Context Manager

Question: Can you create a reentrant context manager?

Answer:

```
from contextlib import contextmanager
import threading

@contextmanager
def reentrant_lock():
    lock = threading.RLock()  # Reentrant lock
    lock.acquire()
    try:
        yield lock
    finally:
        lock.release()

# Usage
with reentrant_lock() as lock:
    with lock:  # Can enter again
        print("Nested context")
```

Q5: contextlib.ExitStack

Question: What's `contextlib.ExitStack` used for?

Answer: It's used for managing multiple context managers dynamically:

```
from contextlib import ExitStack

with ExitStack() as stack:
    files = [
        stack.enter_context(open(f'file{i}.txt'))
        for i in range(5)
    ]
    # All files will be properly closed when exiting
```

Q6: Testing Context Managers

Question: How do you test context managers?

Answer:

```
import unittest
from unittest.mock import MagicMock

class TestContextManager(unittest.TestCase):
    def test_context_manager(self):
        mock_resource = MagicMock()

        class TestCM:
            def __enter__(self):
                return mock_resource
            def __exit__(self, *args):
                mock_resource.cleanup()

        with TestCM() as resource:
            resource.do_something()

        mock_resource.do_something.assert_called_once()
        mock_resource.cleanup.assert_called_once()
```

Built-in Context Managers

- `open()` - File operations
- `threading.Lock()` - Thread synchronization
- `decimal.localcontext()` - Decimal precision
- `tempfile.TemporaryDirectory()` - Temporary directories
- `unittest.mock.patch()` - Testing mocks

- `contextlib.suppress()` - Suppress specific exceptions
- `contextlib.redirect_stdout()` - Redirect stdout
- `contextlib.redirect_stderr()` - Redirect stderr

Advanced Examples

Transaction Management

```
@contextmanager
def transaction(connection):
    cursor = connection.cursor()
    try:
        yield cursor
        connection.commit()
    except Exception:
        connection.rollback()
        raise
    finally:
        cursor.close()

# Usage
with transaction(db_connection) as cursor:
    cursor.execute("INSERT INTO users VALUES (?)", (user,))
```

Temporary File with Cleanup

```
@contextmanager
def temp_file(filename):
    import os
    try:
        f = open(filename, 'w')
        yield f
    finally:
        f.close()
        if os.path.exists(filename):
            os.remove(filename)

# Usage
with temp_file('temp.txt') as f:
    f.write("Temporary data")
```

Timing Context Manager

```
import time
from contextlib import contextmanager
```

```python
@contextmanager
def timing(label="Operation"):
    start = time.perf_counter()
    try:
        yield
    finally:
        end = time.perf_counter()
        print(f"{label}: {end - start:.4f} seconds")

# Usage
with timing("Database query"):
    # Your code here
    time.sleep(0.5)
```

Change Directory Context Manager

```python
import os
from contextlib import contextmanager

@contextmanager
def change_dir(destination):
    original_dir = os.getcwd()
    try:
        os.chdir(destination)
        yield
    finally:
        os.chdir(original_dir)

# Usage
with change_dir('/tmp'):
    # Working in /tmp directory
    print(os.getcwd())
# Back to original directory
```

Key Benefits

- **Resource Cleanup:** Automatic cleanup even if exceptions occur
- **Exception Safety:** Proper handling of exceptions during resource management
- **Code Clarity:** Clear entry and exit points for resource usage
- **Memory Management:** Prevention of resource leaks
- **Deterministic Behavior:** Guaranteed cleanup operations

Best Practices

1. Always return `False` or `None` from `__exit__()` unless you explicitly want to suppress exceptions
2. Use `@contextmanager` decorator for simple context managers
3. Ensure cleanup code is in the `finally` block when using `@contextmanager`
4. Don't forget to return the resource from `__enter__()` if you want to use it with `as`
5. Test both success and exception paths
6. Document whether your context manager is reentrant or thread-safe

---

Common Pitfalls

Wrong: Forgetting cleanup in exception cases

```
@contextmanager
def bad_example():
    resource = acquire()
    yield resource
    release(resource)  # Won't run if exception occurs!
```

Correct: Using try-finally

```
@contextmanager
def good_example():
    resource = acquire()
    try:
        yield resource
    finally:
        release(resource)  # Always runs!
```

Quick Reference

```
# Basic pattern
with context_manager() as resource:
    # Use resource
    pass

# Multiple managers
with cm1() as r1, cm2() as r2:
    pass

# Suppressing exceptions
from contextlib import suppress
with suppress(FileNotFoundError):
```

```
    os.remove('file.txt')

# Nested contexts
with outer():
    with inner():
        pass
```

Pro Tip: Context managers are one of Python's most elegant features for resource management. Master them to write cleaner, safer code!