What is context managers in python?

`In python`, Context managers are a powerfull features that helps manage resources efficently, ensuring that setup and clean up tasks handled gracefully. They are most commonly used with the `with` statment to manage resources like `files`, `network connections` or `database connections`.

A context manager is an object that defines methods for setting up and tearing down a context for a block of code.

`Methods:`

- `__enter__:` This method is invoked when the `with` block is entered. The value returned by `__enter__` is assigned to the variables after the `as` keyword in the `with` statement.
- `__exit__:` This method is invoked when the `with` block is exited. If `True`, the exception is suppressed. If `False` (or if omitted), the exception is propagated.

  `Parameters:`

  - `exc_type:` The exception type, if an exception occured.
  - `exc_value:` The exception value, if an exception occured.
  - `exc_tb:` The traceback object, if an exception occured.

`Example:`

```
with open('file.md') as file:
    data = file.read()
    print(data)

print(file.closed) # print True
```

`open('file.md')` is a context manager.

`Create custom context manager:`

`Example:1`: Stream class for file management

```
class Stream:
    def __init__(self, path: str, mode: str = 'r') -> None:
        self.path = path
        self.mode = mode
        self.filestream = None

    def __enter__(self):
        self.filestream = open(self.file, self.mode)
        return self.filestream

    def __exit__(self, exc_type, exc_value, traceback):
        if exc_type is not None:
            print(f"Exception type : {exc_type}")
            print(f"Exception value : {exc_value}")
            print(f"Exception traceback : {traceback}")
        self.filestream.close()


# use
with Stream("file.md", "r") as file:
    d = file.read()
    print(d)
```

`Example: 2` Safe exception handling with divide by zero

```
class SafeDivide:
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is ZeroDivisionError:
            print("Division by zero is not allowed")
            return True  # Suppress the exception
        return False  # Propagate other exceptions


with SafeDivide() as sd:
    result = 1 / 0  # This will be handled by __exit__
```

`Example: 3` Network connections

```
import requests

class SessionManager:
    def __enter__(self):
        self.session = requests.Session()
        return self.session

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.session.close()

with SessionManager() as session:
    response = session.get('https://api.example.com/data')
    print(response.json())
```

Or create via `contextlib` module

`Example:`

```python
# contextlib module
from contextlib import contextmanager

@contextmanager
def stream(path: str, mode: str = "r"):
    file = open(path, mode)
    yield file
    file.close()

with stream("file.md", "r") as file:
    d = file.read()
    print(d)
```

`Context Managers in Asynchronous Programming:`

Python’s `async` and `await` keywords support asynchronous context managers, which are used with the `async with` statement.

`Creating an Asynchronous Context Manager:`

```python
class AsyncResource:
    async def __aenter__(self):
        print("Acquiring resource asynchronously")
        return "Async Resource"

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        print("Releasing resource asynchronously")
        return False  # Propagate exception

# Usage with `async with`
async def async_main():
    async with AsyncResource() as resource:
        print(f"Using {resource}")

# Run the async main function
import asyncio
asyncio.run(async_main())
```

`Chaining Context Managers:`

```python
with open('file1.txt') as file1, open('file2.txt') as file2:
    content1 = file1.read()
    content2 = file2.read()
```

`Context Managers for Locks:` Context managers are commonly used for thread synchronization with locks:

```python
import threading

class ThreadSafeResource:
    def __init__(self):
        self.lock = threading.Lock()

    def __enter__(self):
        self.lock.acquire()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.lock.release()

# Usage
resource = ThreadSafeResource()
with resource:
    # Critical section
    pass
```

`Context Managers for Temporary Directories:`

```python
import tempfile
import shutil

class TemporaryDirectory:
    def __enter__(self):
        self.dir = tempfile.mkdtemp()
        return self.dir

    def __exit__(self, exc_type, exc_val, exc_tb):
        shutil.rmtree(self.dir)

with TemporaryDirectory() as temp_dir:
    print(f"Temporary directory created at {temp_dir}")
```