Context_Manager

May 7, 2021

1 Context Manager

Is an object design to be used in a `with` statement. It ensures that resources are properly and automatically managed. A Context Manager object needs to implement two methods `__enter__`, `__exit__`.

- `__enter__`: prepares the manager for use. For example acquires the resource.
- `__exit__`: cleans up the manager. For example releases the resource.

```
with expression as x:
    body
```

The value of `expression.__enter__(self)` is bound to `x`, not the value of the expression. It can return values of anytime, commonly it returns the context manager itself.

The method `expression.__exit__(self, exc_type, exc_val, exc_tb)` get always called. Even when an exception is raised, it receives the exception type, value and traceback. If `__exit__` return `False` the exception is propagated. Remember that by default functions return `None` and `None` evaluates to `False`. `__exit__` should **never** explicitly re-raise exceptions. It should **only** raise exceptions if it fails itself.

[1]:
```
class MyContextManager:
    def __enter__(self):
        print('__enter__')
        return 'Hello darkness my old friend'

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f'__exit__({exc_type},{exc_val},{exc_tb})')
        return
```

[3]:
```
with MyContextManager() as mcm:
    print(mcm)
```

```
__enter__
Hello darkness my old friend
__exit__(None,None,None)
```

[4]:
```
with MyContextManager() as mcm:
    raise ValueError('Wololo')
    print(mcm)
```

```
__enter__
__exit__(<class 'ValueError'>,Wololo,<traceback object at 0x1052eef00>)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-a4e6c3e07cad> in <module>
      1 with MyContextManager() as mcm:
----> 2     raise ValueError('Wololo')
      3     print(mcm)

ValueError: Wololo
```

[7]:
```
f = open('wololo', 'w')
with f as g:
    print(f is g)
```

```
True
```

1.1 Contextlib

Contextlib is a standard library module that for working with context managers that provides common utilities for tasks involving the with statement.

`contextlib.contextmanager` is a decorator you can use to create new context managers.

```
@contextlib.contextmanager
def my_context_manager():
    try:
        # Prepare resource and return it with yield (__enter__)
        yield value
        # Normal release of the resource (__exit__) when there is no exception
    except:
        # Section handling (__exit__) when an exception has been raised exception
        raise
```

It uses standard exception handling to propagate exceptions. Explitcitly re-raise - or don't catch - to propagate exceptions. Swallows exceptions by not re-raising them.

[9]:
```
import contextlib

@contextlib.contextmanager
def my_context_manager():
    print('my_context_manager: enter')
    try:
        yield 'You are in the with block'
        print('my_context_manager: exit (no exception)')
    except:
        print('my_context_manager: exit (exception)')
```

[10]:
```
with my_context_manager() as x:
    print(x)
```

```
my_context_manager: enter
You are in the with block
my_context_manager: exit (no exception)
```

[11]:
```
with my_context_manager() as x:
    raise ValueError('Wololo')
    print(x)
```

```
my_context_manager: enter
my_context_manager: exit (exception)
```

1.2 Multiple context managers

Context managers can be stack one of top of the other.

```
with cm1() as a:
    with cm2() as b:
        BODY
```

is the same as

```
with cm1() as a, cm2() as b:
    BODY
```

[12]:
```
@contextlib.contextmanager
def simple(name):
    print(f'entering {name}')
    yield name
    print(f'exiting {name}')
```

[14]:
```
with simple('a') as a, simple('b') as b:
    pass
```

```
entering a
entering b
exiting b
exiting a
```