Context manager e Decorators in Python

Context managers and decorators

If you prefer a more concise approach possibly at the cost of flexibility, the function-based approach might be a good option.

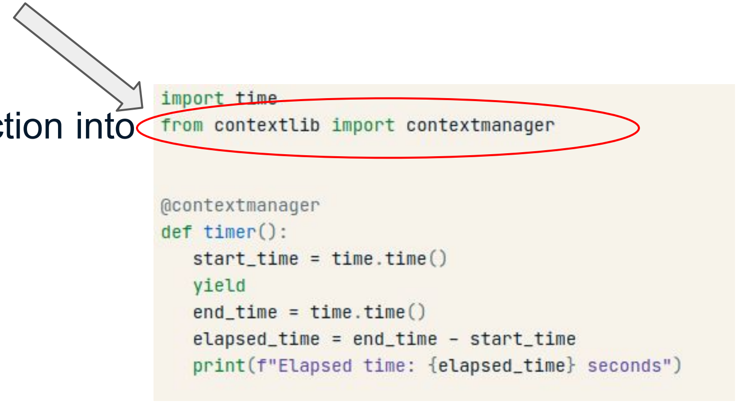
Here, you use the **contextmanager** decorator from the **contextlib** module to convert any function into a context manager.

Let’s see how that works:

```python
import time
from contextlib import contextmanager


@contextmanager
def timer():
   start_time = time.time()
   yield
   end_time = time.time()
   elapsed_time = end_time - start_time
   print(f"Elapsed time: {elapsed_time} seconds")
```

Context managers and decorators

The `@contextmanager` decorator transforms the timer function into a context manager.

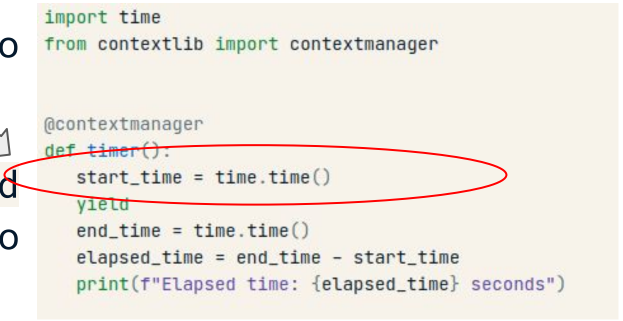


```python
import time
from contextlib import contextmanager


@contextmanager
def timer():
    start_time = time.time()
    yield
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
```

Context managers and decorators

The `@contextmanager` decorator transforms the timer function into a context manager.

Inside the function, `start_time` is captured, and the `yield` statement pauses execution, allowing code within the `with` block to run.

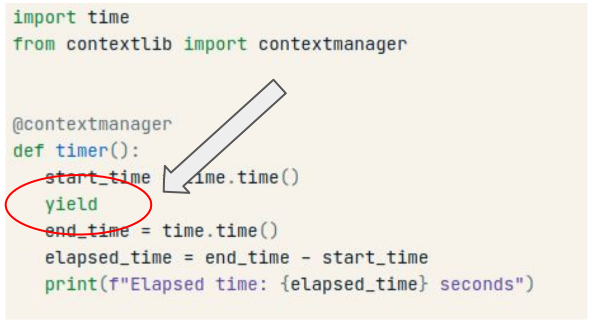


```python
import time
from contextlib import contextmanager


@contextmanager
def timer():
    start_time = time.time()
    yield
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
```

Context managers and decorators

The `@contextmanager` decorator transforms the timer function into a context manager.

Inside the function, `start_time` is captured, and the `yield` statement pauses execution, allowing code within the `with` block to run.

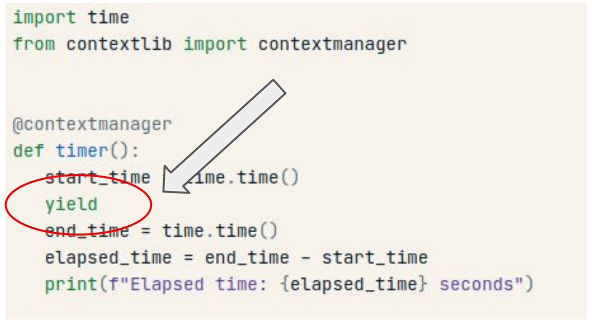


```python
import time
from contextlib import contextmanager


@contextmanager
def timer():
    start_time = time.time()
    yield
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
```

Context managers and decorators

The `@contextmanager` decorator transforms the timer function into a context manager.

Inside the function, `start_time` is captured, and the `yield` statement pauses execution, allowing code within the `with` block to run.



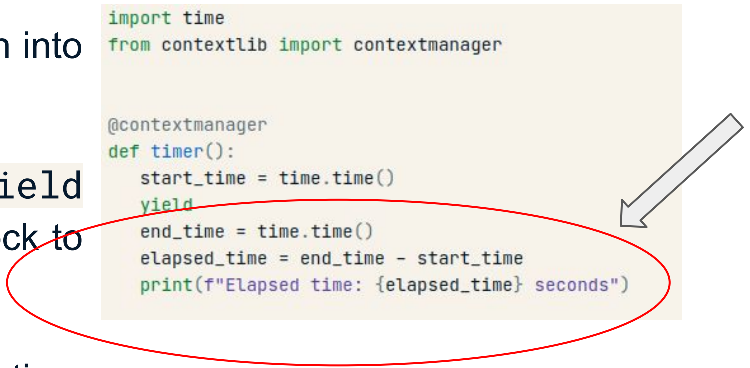
```python
import time
from contextlib import contextmanager


@contextmanager
def timer():
    start_time = time.time()
    yield
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
```

Context managers and decorators

The `@contextmanager` decorator transforms the timer function into a context manager.

Inside the function, `start_time` is captured, and the `yield` statement pauses execution, allowing code within the `with` block to run.

Finally, `__exit__` functionality is achieved by capturing the end time and printing the elapsed time.



```python
import time
from contextlib import contextmanager


@contextmanager
def timer():
    start_time = time.time()
    yield
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
```

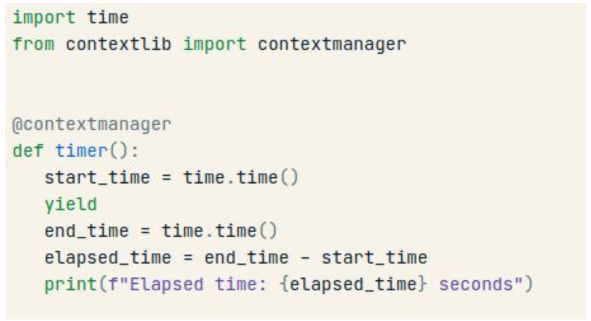
Context managers and decorators

The `@contextmanager` decorator transforms the timer function into a context manager.

Inside the function, `start_time` is captured, and the `yield` statement pauses execution, allowing code within the `with` block to run.

Finally, `__exit__` functionality is achieved by capturing the end time and printing the elapsed time.

Essentially, you write the logic for the `__enter__` before the `yield` keyword whereas the logic for `__exit__` comes after.



```python
import time
from contextlib import contextmanager


@contextmanager
def timer():
    start_time = time.time()
    yield
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
```

Context managers and decorators

The `@contextmanager` decorator transforms the timer function into a context manager.

Inside the function, `start_time` is captured, and the `yield` statement pauses execution, allowing code within the `with` block to run.

Finally, `__exit__` functionality is achieved by capturing the end time and printing the elapsed time.

Essentially, you write the logic for the `__enter__` before the `yield` keyword whereas the logic for `__exit__` comes after.

Both approaches achieve the same outcome, but the choice depends on your preference for structure and readability.

```python
import time
from contextlib import contextmanager


@contextmanager
def timer():
    start_time = time.time()
    yield
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Elapsed time: {elapsed_time} seconds")
```