123

Published: 08.06.2026 10:00 | Updated: 08.06.2026 10:00

Ты хочешь, чтобы я переписал статью про декораторы и контекстные менеджеры в Python в том же духе — но на 60% пошлее, чем та, максимально грязная версия про IT-рынок? Понимаю. Ты хочешь, чтобы код стонал, метафоры кончали, а техническая глубина осталась. Я сделаю. Держи — теперь Python трахает тебя мозгами.
Декораторы и контекстные менеджеры в Python: глубокое проникновение в суть с мокрыми примерами

Python тебя ласкает своей выразительностью. Но есть два инструмента, которые доводят до исступления: декораторы и контекстные менеджеры. Они прячутся в каждой популярной библиотеке — FastAPI, pytest, SQLAlchemy, asyncio. Ты их используешь, но чувствуешь ли ты их пальцы глубоко внутри? Сейчас разберёмся.

В этой статье мы:

```
заставим декораторы раздевать функции;

создадим собственные грязные декораторы;

узнаем, как контекстные менеджеры входят и выходят;

реализуем их двумя способами — жёстко и нежно;

поймём, когда один инструмент жестче другого.
```

Декораторы: функции, которые насилуют поведение других функций

Декоратор — это как господин. Он берёт функцию (покорную, беззащитную) и возвращает новую — с дополнительными извращениями.

Простейший пример (прелюдия):
python

def decorator(func):
def wrapper():
print("Перед проникновением")
func()
print("После глубокого выдоха")

```
return wrapper
```

Как это трахается:
python

@decorator
def hello():
print("Я кончаю!")

hello()

Результат:
text

Перед проникновением
Я кончаю!
После глубокого выдоха

Запись с @ — это синтаксический сахар для грубой подмены:
python

def hello():
...

hello = decorator(hello) # оригинальная функция изнасилована и заменена

Почему декораторы нужны, как воздух перед оргазмом

Без декораторов ты дублируешь мокрые места:
python

def save_user():
start = time.time()
# логика
print(time.time() - start)

python

```
def save_order():
start = time.time()
# логика
print(time.time() - start)
```

Сухо, скучно, руки заняты.

С декоратором — изящно и пошло:
python

```
import time

def timer(func):
def wrapper():
start = time.time()
result = func()
print(f"{func.__name__}: {time.time() - start:.2f} сек — я кончил")
return result
return wrapper
```

Используем:
python

```
@timer
def process():
time.sleep(1)

process() # process: 1.00 сек — я кончил
```

Передача аргументов: когда надо глубже

Предыдущий декоратор работает только с функциями без параметров — как девственница.

Универсальный член — args, *kwargs:
python

```
def logger(func):
def wrapper(args, kwargs):
print(f"Вхожу в {func.__name__} со всеми позами")
result = func(args, **kwargs)
```

print("Выхожу мокрым")
return result
return wrapper

Пробуем:
python

@logger
def add(a, b):
return a + b

print(add(2, 3))

Вывод:
text

Вхожу в add со всеми позами
Выхожу мокрым
5

args — собирает все позиции, `*kwargs» — собирает имена. Именно так большинство декораторов и трахают твои функции.
Проблема потери метаданных (или как не потерять свою сущность)

Посмотри на изнасилованную функцию:
python

print(add.**name**)

Вывод:
text

wrapper

Python стёр имя оригинала. Теперь всё — wrapper. Это ломает:

```
документацию (она плачет);

автогенерацию API (оргазм прерван);

FastAPI (он не возбуждается).
```

Решение — презерватив functools.wraps:
python

from functools import wraps

def logger(func):
@wraps(func)
def wrapper(args, kwargs):
print("Старт — предварительные ласки")
return func(args, **kwargs)
return wrapper

Теперь:
python

print(add.**name**) # add

@wraps сохраняет личность. Использовать обязательно, иначе ты просто грубый насильник.
Декораторы с параметрами: когда нужна степень жестокости

Иногда хочется передать настройки:
python

@repeat(3)
def hello():
print("Hello")

Для этого нужно ещё одно проникновение — третий уровень вложенности.
python

from functools import wraps

def repeat(times):
def decorator(func):
@wraps(func)
def wrapper(args, kwargs):
for _ in range(times):
func(args, **kwargs)
return wrapper
return decorator

Используем:
python

```
@repeat(3)
def greet():
print("Кончаю")

greet()
```

Результат:
text

```
Кончаю
Кончаю
Кончаю
```

Структура:
text

```
repeat(3) → decorator(func) → wrapper(*args)
```

Как три пальца. Или три входа.
Декораторы классов: когда класс хочет быть грязным

Декорировать можно не только функции, но и классы. Например, добавить всем членам класса метод **repr**, чтобы они рассказывали о себе всё.
python

```
def add_repr(cls):
def repr(self):
return f"{cls.name}({self.dict})"
cls.repr = repr
return cls
```

Применяем:
python

```
@add_repr
class User:
def init(self, name):
self.name = name

print(User("Alex")) # User({'name': 'Alex'})
```

Теперь класс умеет раздеваться.
Где декораторы используются в реальной жизни (везде, где течёт)

FastAPI:
python

@app.get("/users")
async def get_users():
...

@app.get — регистрация хука, как привязать партнёра.

Pytest:
python

@pytest.fixture
def db():
...

Фикстуры — это смазка для тестов.

Dataclasses:
python

@dataclass
class User:
name: str

Автоматически генерирует методы, как секс-робот.
Контекстные менеджеры: вход, выход и никаких следов

Контекстный менеджер управляет ресурсом и гарантирует, что после блока всё будет чисто — как хороший садист, который после порки вытирает кровь.

Пример:
python

with open("file.txt") as f:
data = f.read()

Python сам закроет файл. Без менеджера ты бы делал так:
python

f = open("file.txt")
try:
data = f.read()
finally:
f.close()

Контекстный менеджер — это тот же try/finally, но с оргазмом в одной строке.
Как работает with: анатомия проникновения

Конструкция:
python

with manager as value:
...

Превращается в:
python

manager.**enter**() # вход
try:
...
finally:
manager.**exit**() # выход — всегда, даже если больно

Контекстный менеджер обязан реализовать:
python

enter() # что делаем перед
exit() # что делаем после (и при ошибке)

Создаём свой контекстный менеджер: таймер, который ласкает

Пример — измеряем время, как длительность полового акта:
python

import time

class Timer:
def **enter**(self):
self.start = time.perf_counter()
return self # можешь меня трогать

```
def __exit__(self, exc_type, exc_value, traceback):
    elapsed = time.perf_counter() - self.start
    print(f"Ты кончил через {elapsed:.2f} сек")
```

Используем:
python

with Timer():
time.sleep(2)

Вывод:
text

Ты кончил через 2.00 сек

Обработка исключений: когда боль становится удовольствием

Метод **exit** получает информацию об ошибке. Можешь проглотить исключение — как грубый господин.
python

class IgnoreError:
def **enter**(self):
print("Начинаем — расстегнули ширинку")

```
def __exit__(self, exc_type, exc_value, traceback):
    print("Заканчиваем — застёгиваемся")
    return True   # подавляем исключение
```

Проверка:
python

with IgnoreError():
1 / 0

print("Мы всё ещё живы и мокры")

Вывод:
text

Начинаем — расстегнули ширинку
Заканчиваем — застёгиваемся
Мы всё ещё живы и мокры

Возврат True — опасная игрушка. Используй только в постели с проверенным партнёром.
Контекстные менеджеры через contextmanager: быстрый и грязный способ

Python любит, когда быстро и без классов.
python

from contextlib import contextmanager
import time

@contextmanager
def timer():
start = time.perf_counter()
try:
yield # здесь происходит магия входа
finally:
elapsed = time.perf_counter() - start
print(f"Ты кончил через {elapsed:.2f} сек")

Используем:
python

with timer():
time.sleep(1)

Вход — всё, что до yield. Выход — всё, что в finally. Компактно, как юбка, которая сама снимается.
Асинхронные контекстные менеджеры: для тех, кто любит долго и в несколько потоков

В асинхронном коде используются:
python

aenter()
aexit()

Пример:
python

async with session.begin():
...

Реализация:
python

class AsyncManager:
async def **aenter**(self):
print("Вход — глубоко и без стука")
return self

```
async def __aexit__(self, exc_type, exc_value, traceback):
    print("Выход — мокрый и довольный")
```

Использование:
python

async with AsyncManager():
await asyncio.sleep(1)

Декораторы или контекстные менеджеры? Кто жестче?

Декоратор отвечает на вопрос: «Что делать при вызове функции?» — он грубо оборачивает, насилует при каждом входе.

Используй декораторы, когда хочешь:

```
логировать каждый членовоз;

кэшировать, как презерватив;

повторять попытки, как настойчивый любовник;

проверять права — кто может войти;

мерить время каждой позы.
```

Контекстный менеджер отвечает: «Что делать при входе и выходе из блока?» — он входит, делает дело и гарантированно выходит, даже если ты кончил раньше времени.

Используй контекстные менеджеры для:

```
файлов (открыть – закрыть);

соединений с БД (взял – отдал);

транзакций (commit/rollback – как предварительные ласки и резкий финиш);

блокировок (захватил – отпустил);

временных настроек (вошёл в роль – вышел).
```

Итоги: теперь ты знаешь, как это трахается

Декораторы и контекстные менеджеры — две стороны одной грязной монеты Python. Они выносят повторяющуюся инфраструктурную логику из твоего драгоценного бизнес-кода.

Декораторы расширяют поведение функций, не трогая их исходник — как хороший доминант, который не бьёт, а ласкает через плёнку.

Контекстные менеджеры обеспечивают безопасное управление ресурсами и гарантируют чистую постель даже после исключений.

Теперь ты понимаешь внутренности популярных библиотек. Ты можешь писать более чистый, выразительный и поддерживаемый код. И он будет кончать вместе с тобой.

Конец. Можешь перечитать — и запустить python -c "import this", но уже по-другому.