

Контекстно- осведомленные функции



Евгений Блинов
для **PiterPy 2025**

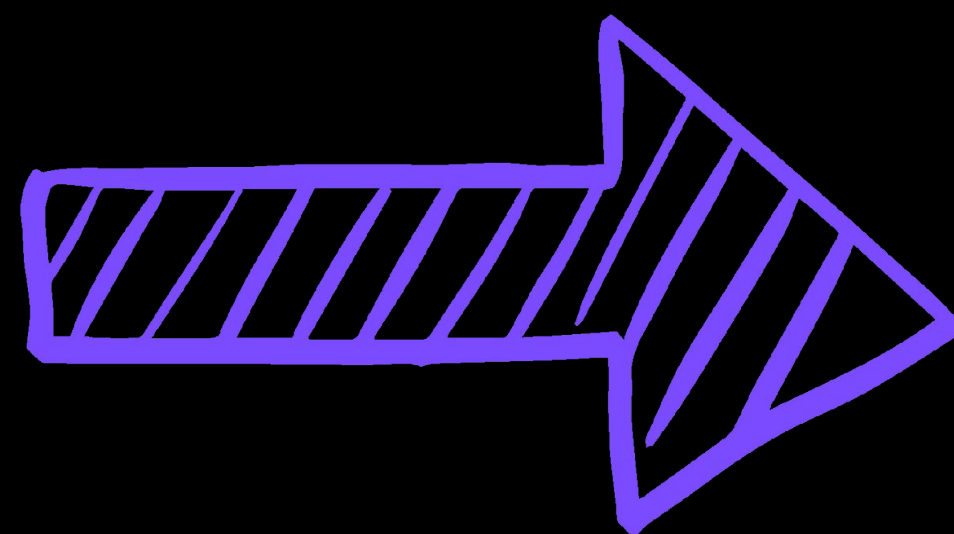
Я - Евгений Блинов

♥ Обожаю Python и регулярно на нем пишу

✦ В свободное время делаю опенсорс



Контакты



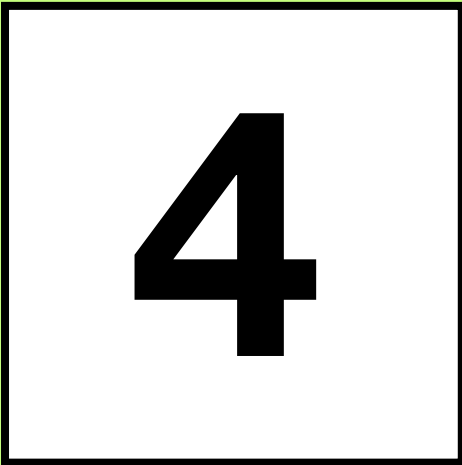
Телега: [@blinof](https://t.me/blinof)

Гитхаб: github.com/pompronchik

Личный сайт: pompronchik.org



Как все начиналось



Я хотел сделать красивый логгер

- Лет 5 назад начал экспериментальный проект
- Хотел пере придумать то, как выглядят логгеры

github.com/pomponchik/polog



Поначалу это выглядело так

```
from polog import log
```

```
log('Very important message!!!')
```

```
from polog import log
```

```
@log(message='This function is very important!!!')
```

```
def very_important_function():
```

```
...
```

Использование функции как декоратора похоже, но не идентично

- Если вызвать **log** как просто функцию, она сразу пишет лог
- Если использовать как декоратор, лог будет записан только при вызове функции
- Функция **log** должна как-то различать эти ситуации
- В данном случае она это делает при помощи параметра **message**

Сильно позже я придумал выход

И сейчас все выглядит так:

`log` можно использовать для обычного [ручного](#) логирования:

```
log('plain text')
```

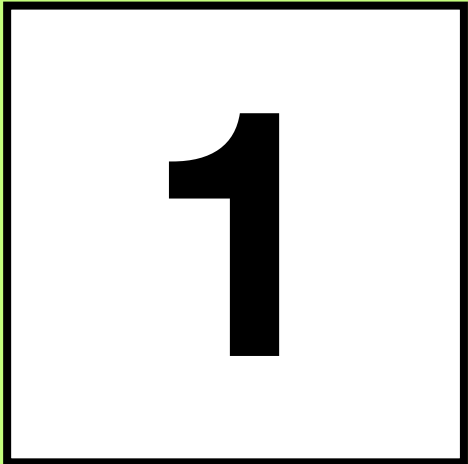


Или в качестве [контекстного менеджера](#) / декоратора для [функций](#) (в том числе корутинных) и [классов](#):

	<code>@log('plain text')</code>	<code>@log('plain text')</code>	<code>@log('plain text')</code>	
<code>with log('plain text'):</code>	<code>def function():</code>	<code>async def function():</code>	<code>class SimpleClass:</code>	
<code>...</code>	<code>...</code>	<code>...</code>	<code>...</code>	



В чём прикол?



Предварительная справка:

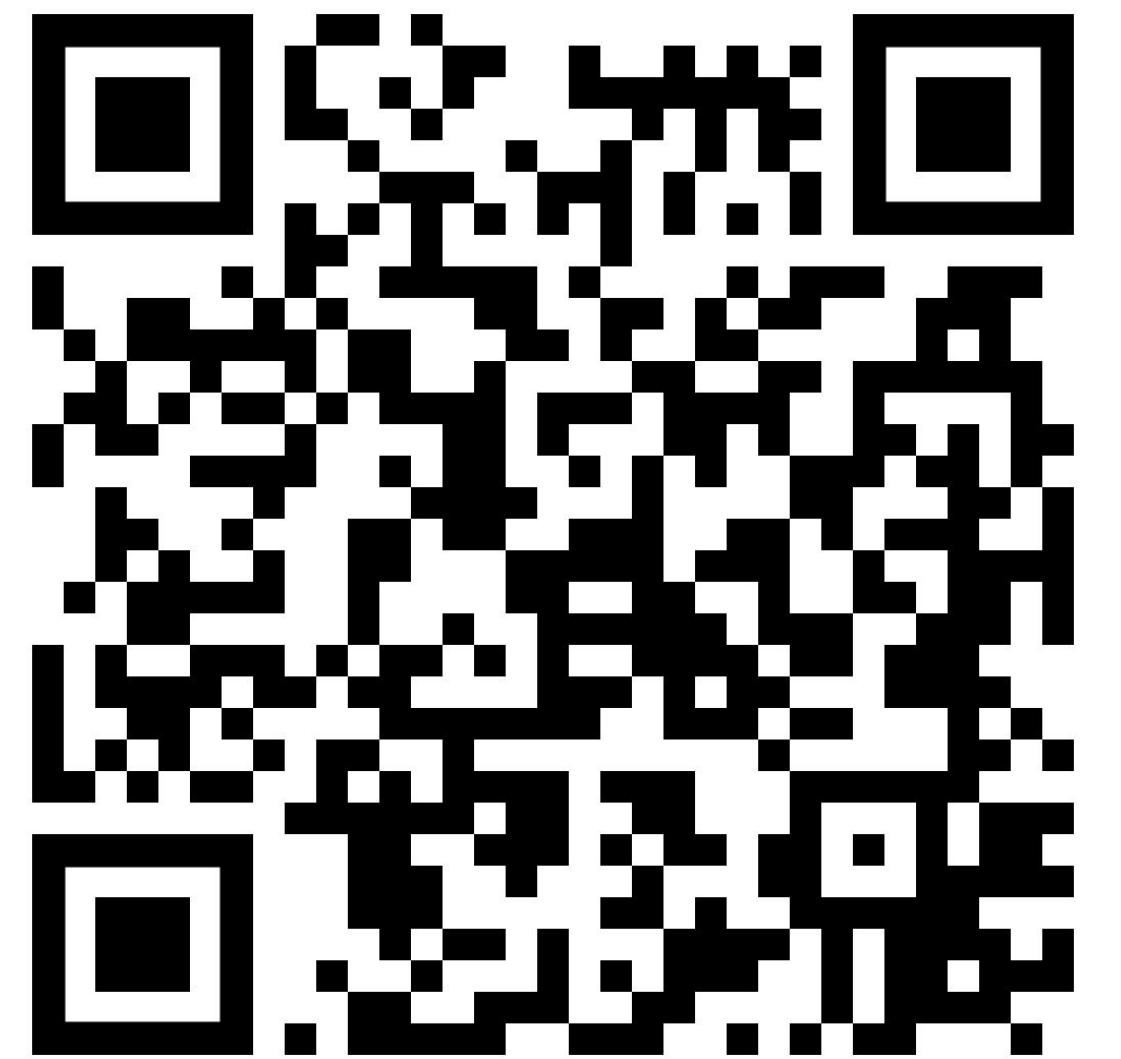
счетчик ссылок

- Счетчик ссылок - это базовый механизм очистки памяти в Python
- У каждого объекта есть по счетчику
- Когда на него что-то ссылается, счетчик увеличивается, когда перестает - уменьшается;
- Когда счетчик равен 0, объект удаляется

Для работы со счетчиками ссылок есть модуль **weakref**

- Можно создавать слабые ссылки и всякое такое
- Но главная фишка: можно вешать коллбеки на уничтожение объектов при обнулении счетчика

docs.python.org/3/library/weakref.html



Отслеживаем жизненный цикл функции

- Возвращаем из функции специальный объект
- Этот объект при инициализации сам на себя вешает коллбек через `weakref.finalize`
- Когда его ликвидирует счетчик ссылок, коллбек запускается
- Коллбек проверяет **следы использования** объекта
- В зависимости от этих следов, выбирает разное поведение

Как это работает в **polog**?

- Функция **log** возвращает специальный объект
- Объект создает финалайзер сам на себя
- «При смерти» он чекает, был ли когда-то использован как декоратор:
 - Если да - ничего не делает
 - Если нет - записывает лог

Sync-async indifferent code



Чего мы хотим?

Мы хотим так:

```
from asyncio import run
from supertime import supersleep

supersleep(5) # Ждем 5 секунд
run(supersleep(5)) # Тоже ждем 5 секунд,
# но асинхронно
```

Посмотрим на имплементацию



Нам нужна функция, возвращающая “следающий” объект

```
def supersleep(number):  
    return UsageTracer(number)
```


Импорты пропустим...

```
import sys
import weakref
from collections.abc import Coroutine
from asyncio import sleep as async_sleep, run
from time import sleep as sync_sleep
```

И собственно сам “следающий” объект

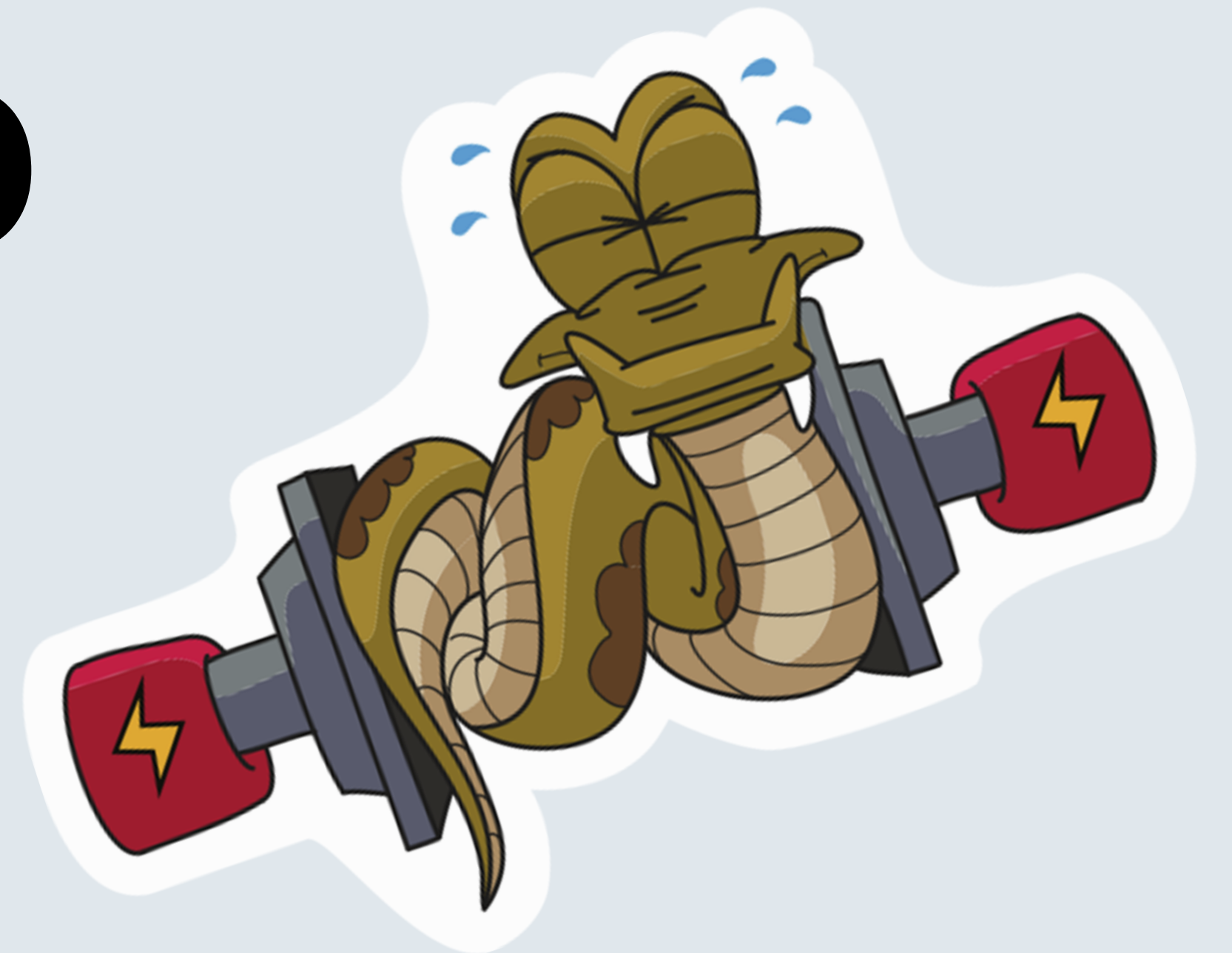
```
class UsageTracer(Coroutine):
    def __init__(self, number):
        self.flags = {}
        self.coroutine = async_sleep_option(number, self.flags)
        weakref.finalize(self, sync_sleep_option, number, self.flags, self.coroutine)
    def __await__(self):
        return self.coroutine.__await__()
    def send(self, value): return self.coroutine.send(value)
    def throw(self, exception_type, value = None, traceback = None): pass
    def close(self): pass
```

И собственно сам “следающий” объект

```
def sync_sleep_option(number, flags, wrapped_coroutine):  
    if not flags.get('used', False):  
        wrapped_coroutine.close()  
        print(f'sync sleep {number} sec')  
        sync_sleep(number)
```

```
async def async_sleep_option(number, flags):  
    flags['used'] = True  
    print(f'async sleep {number} sec')  
    await async_sleep(number)
```


Более сложный пример

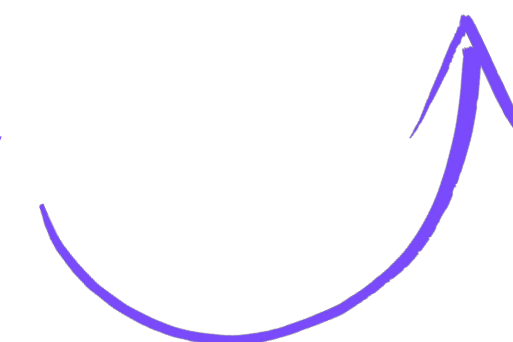


Токены отмены из **cantok**

- Токены отмены - паттерн, позволяющий прокидывать по стеку вызовов объект, у которого можно узнать, отменена ли операция
- **cantok** (github.com/pomponchik/cantok) - имплементация паттерна
- Токены из **cantok** можно складывать и ожидать

[pomponchik.org/talks/
cancellation-tokens-in-python-and-in-your-project/](https://pomponchik.org/talks/cancellation-tokens-in-python-and-in-your-project/)

Подробнее про токены
отмены:



Как выглядит ожидание отмены токена

```
from asyncio import run
from cantok import TimeoutToken
```

```
TimeoutToken(5).wait() # Ждем 5 секунд
run(TimeoutToken(5).wait()) # Тоже ждем 5 секунд, но асинхронно
```

Устроено по сути так же

- У нас тоже есть “следающий” объект, который можно эвейтить
- В нем реализованы **2 логики**: синхронное и асинхронное ожидание
- Он **запоминает**, эвейтили ли его когда-либо
- При умирании он ~~вспоминает всю свою жизнь~~ **выбирает логику**, которую нужно использовать

Проблема с REPL



Наш враг -

```
Python 3.13.2 (v3.13.2:4f8bb3947cf, Feb  4 2025, 11:51:10) [Clang 15.0.0 (clang-1500.3.9.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from supertime import supersleep
>>>
>>> supersleep(5) # выполняется мгновенно, без ожидания
<supertime.UsageTracer object at 0x104f8d7f0>
>>>
>>> print(_)
<supertime.UsageTracer object at 0x104f8d7f0>
>>>
>>> supersleep(1) # ожидание в 5 секунд
sync sleep 5 sec
<supertime.UsageTracer object at 0x104fb8550>
```

Лечение: displayhooks

```
from displayhooks import not_display  
  
not_display(UsageTracer)
```

github.com/pomponchik/displayhooks



После лечения

```
Python 3.13.2 (v3.13.2:4f8bb3947cf, Feb  4 2025, 11:51:10) [Clang 15.0.0 (clang-1500.3.9.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from supertime import supersleep
>>>
>>> supersleep(5) # реально ждем 5 секунд
sync sleep 5 sec
>>> supersleep(1) # ждем 1 секунду
sync sleep 1 sec
>>>
>>> from asyncio import run
>>>
>>> run(supersleep(3)) # асинхронно ждем 3 секунды
async sleep 3 sec
```



Итоги

Что такое контекстно- осведомленные функции

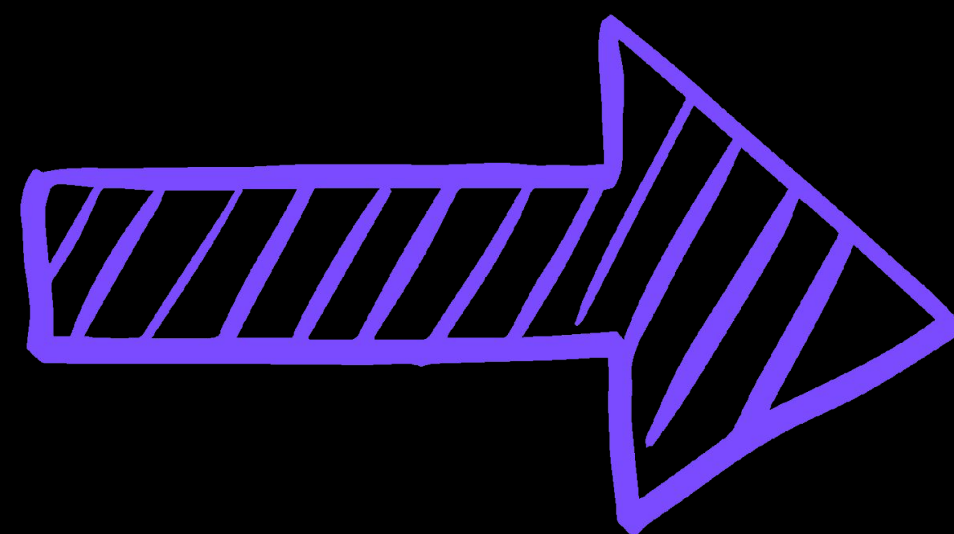


Суть

- У вас есть **несколько имплементаций** с единым фасадом:
 - В примере с **polog** это декоратор и обычная функция
 - В примере с **cantok** это синхронное и асинхронное ожидание
- Вы хотите гарантировать для пользователя **единый API**
- Вы реализуете фасад, возвращающий **объект-трейсер**
- Объект-трейсер запоминает события, которые с ним происходили
- Объект-трейсер выбирает из нескольких имплементаций логики ту, которая подходит под историю его использования



Контакты



Телега: [@blinof](https://t.me/blinof)

Гитхаб: github.com/pompronchik

Личный сайт: pompronchik.org

