Gerenciadores de contexto

Live de Python # 302

Roteiro

1. Um passo antes

Um pouco de teoria

2. with e async with

O bloco de contexto

3. O protocolo

enter/exit aenter/aexit

4. contextlib

Simplificando com geradores

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto

55adriano, Albano Maywitz, Alexandre Costa, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Neto, Alysson Oliveira, Andre Makoski, André Oliveira, Andre Paula, Antonio Filho, Apolo Ferreira, Aurelio Costa, Belisa Arnhold, Bernarducs, Bruno Batista, Bruno Bereoff, Bruno Freitas, Bruno Ramos, Bruno Russian, Brunu, Canibasami, Carlos Gonçalves, Carlos Henrique, Carol Souza, Cauã Oliveira, Celio Araujo, Christian Fischer, Claudemir Cruz, Cleiton Fonseca, Curtos Treino, Daniel Aguiar, Daniel Brito, Daniel Bruno, Daniel Souza, Daniel Wojcickoski, Danilo Silva, David Couto, David Frazao, Dh44s, Diego Guimarães, Diego Suhett, Dilan Nery, Edgar, Elias Moura, Elias Soares, Emerson Rafael, Érico Andrei, Esdras, Everton Silva, Ewertonbello, Fabiano Gomes, Fábio Belotto, Fabiokleis, Felipe Adeildo, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Ferrabras, Fichele Marias, Fightorfall, Francisco Aclima, Franklin Sousa, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Glauber Duma, Gnomo Nimp, Grinaode, Guilherme Felitti, Gustavo Suto, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Herian Cavalcante, Hiago Couto, Idlelive, Iuri Kintschev, Ivan Santiago, Ivansantiagojr, Janael Pinheiro, Jean Victor, Jefferson Antunes, Jerry Ubiratan, Jhonata Medeiros, Joarez Wernke, Jonas Araujo, Jonatas Leon, Jonatas_silva11, Jose Barroso, Joseíto Júnior, José Predo), Josir Gomes, Jota_lugs, Jplay, Jrborba, Ju14x, Juan Felipe, Juli4xpy, Juliana Machado, Julio Cesar, Julio Franco, Julio Gazeta, Julio Silva, Kaio Peixoto, Knaka, Leandro Pina, Leandro Vieira, Leonardo Mello, Leonardo Nazareth, Lilian Pires, Lisandro Pires, Lucas Carderelli, Lucas Castro, Lucasgcode, Lucas Mendes, Lucas Nascimento, Lucas Polo, Lucas Schneider, Luciano_ratamero, Luciano Ratamero, Lúcia Silva, Luidduarte, Luis Ottoni, Luiz Duarte, Luiz Martins, Luiz Paula, Luiz Perciliano, Marcelo Araujo, Marcelo Fonseca, Marcio Freitas, Marcos Almeida, Marcos Oliveira, Marina Passos, Mateusamorim96, Matheus Vian, Medalutadorespacialx, Michael Santos, Miley_presidente, Mlevi Lsantos, Murilo Carvalho, Nhambu, Oopaze, Otávio Carneiro, Patrick Felipe, Programming, Pytonyc, Rafael Ferreira, Rafael Fontenelle, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Ramon Lobo, Renan, Renan Sebastião, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Richard Costa, Richard Sousa, Rinaldo Magalhaes, Robsonpiere, Rodrigo Barretos, Rodrigo Santana, Rodrigo Vieira, Rogério Nogueira, Ronaldocostadev, Rui Jr, Samael Picoli, Samanta Cicilia, Santhiago Cristiano, Scrimf00x, Scrimfx, Shinodinho, Shirakawa, Tarcísio Miranda, Tenorio, Téo Calvo, Teomewhy, Thamires Betin, Tharles Andrade, Thedarkwithin, Thiago, Thiago Araujo, Thiago Lucca, Thiago Paiva, Tiago, Tomás Tamantini, Valdir, Varlei Menconi, Vinícius Areias, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vitoria Trindade, Vladimir Lemos, Vonrroker, Williamslews, Willian Lopes, Xxxxxxxxx, Zero! Studio

Obrigado você <3

Um passo antes

Teoria

Gerenciador de contexto

Antes de explorarmos mais a fundo os mecanismos internos do Python. Um gerenciador de contexto é o que usamos quando definimos o bloco **`with`**.

Talvez a forma mais tradicional e vista por todos os níveis seja a manipulação de arquivos com a função embutida **open**:

```
with open('arquivo.txt') as file:
    conteudo = file.read()
```

Sem o **`with`**, precisaríamos escrever algo como:

```
file = open('arquivo.txt')
conteudo = file.read()
file.close()
```

O objetivo desse slide é avisar sobre o que vamos conversar xD

Recursos / Escopos / Contexto

Antes de cairmos no **`with`**, vamos olhar essa instrução de forma mais "clara":

```
file = open('arquivo.txt')
conteudo = file.read()
file.close()
```

Vamos pensar nessa ordem dos acontecimentos:

Recursos / Escopos / Contexto

Antes de cairmos no **`with`**, vamos olhar essa instrução de forma mais "clara":

```
file = open('arquivo.txt')
conteudo = file.read()
file.close()
```

Vamos pensar nessa ordem dos acontecimentos:

- Um recurso é "criado" e "destruído"
- Um nome criado não terá mais utilidade no escopo
- O "contexto" de uso de **`file`** não se expande além disso

Faça e garanta que será desfeito

Podemos entender isso como um padrão de código

- "abre" e "fecha"
- "cria" e "destrói"
- "faz" e "desfaz"
- "aloca" e "desaloca"

Um padrão que toma conta do "ciclo de vida" de algum objeto. Criando um contexto onde uma determinada operação acontece e "desfazendo" esse contexto.

A referência teórica

De alguma forma, ao buscar em livros/papers, não consegui encontrar algum teórico que tenha catalogado e nomeado esse comportamento.

Ele existe em diversas linguagens de programação diferentes:

Feature	Linguagem	O que faz?
`IDisposable` / `using`	C#	Gerencia a liberação de recursos (via `Dispose`)
`defer`	Go / Swift / Zig	Executa algo no fim do escopo
`ensure`	Ruby	Garante execução final (mesmo em exceções)
`with`	Python	Gerencia contexto (abre/fecha recursos)
RAII	C++	Recurso é liberado no fim do escopo
`Drop` Trait	Rust	Define o comportamento ao destruir o valor

Algo específico para gerenciar um **"recurso"** em um **"contexto"** específico de uso.

Embora o termo pra isso seja **"Gerenciador de contexto"**. Algo aqui me assusta... Que raios quer dizer `contexto`?

A nomenclatura

A nomenclatura

No glossário do termo `Contexto` temos algo como:

Este termo tem **diferentes significados dependendo de onde e como ele é usado**. Alguns significados comuns:

1. O estado ou ambiente temporário estabelecido por um `gerenciador de contexto` por meio de uma instrução `with`.

2. A coleção de ligações de chave-valor associadas a um objeto `contextvars.Context` específico e acessadas por meio de objetos `ContextVar`. Veja também `variável de contexto`.

3. Um objeto `contextvars.Context`. Veja também `contexto atual`.

Contexto

Pela explicação anterior, vimos que contexto tem significados polissêmicos. Ou seja, uma palavra que tem diversos significados dependendo do uso.

Um exemplo de polissemia:

- Eu adoro comer `manga`
- Juliapix usa `manga` longa

Ou seja, a palavra `contexto` em python vai depender de onde ela está sendo usada. Não existe uma resolução geral.

Contexto

Bom... Mas, com a definição de `contexto` talvez consigamos ir mais longe:

> Conjunto de circunstâncias que envolvem um fato e são imprescindíveis para o entendimento deste.

Michaelis

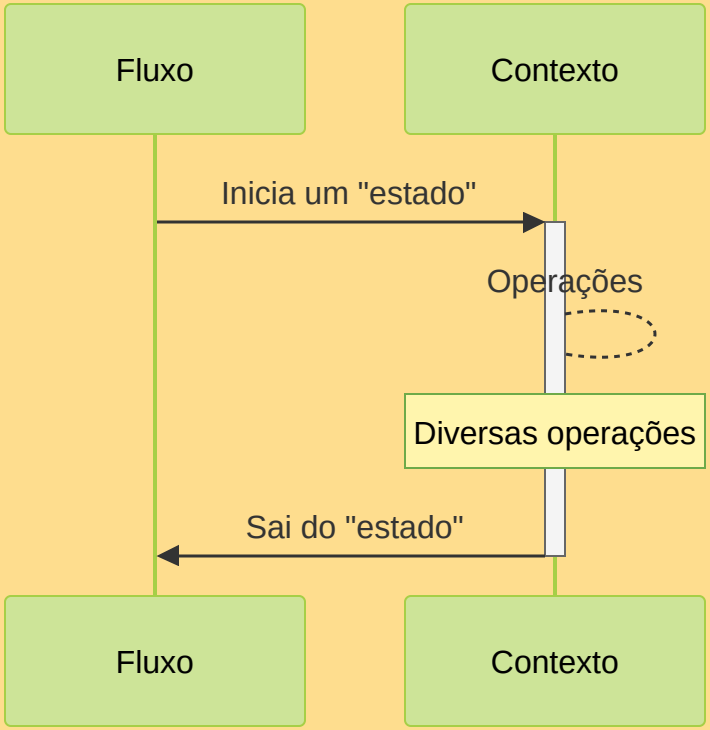
Ou seja, um gerenciador de contexto é algo que controla o ambiente e as condições em que uma operação ocorre. Garantindo que os recursos sejam corretamente inicializados e liberados.

```
with open('arquivo.txt') as file:
    conteudo = file.read()
```

A circunstância em que o arquivo **foi aberto e será fechado após** o seu manejo.

Gerenciador de contexto

Para tentarmos levar isso de forma mais "pragmática". Podemos pensar em estados:



PEP 343 – The “with” Statement

O bloco `with` foi proposto como um mecanismo alternativo ao `try/finally` para a versão 2.5. Algo como:

```
try:
    # faça algo
finally:
    # desfaça aquilo que fez
```

No sentido de algo ia ser "construído" e "destruído" após o "contexto" de execução.

Além disso, define o protocolo de "gerenciamento de contexto". Para os objetos que podem ser usados pelo `with`.

Um exemplo à moda antiga

Usando `try/finally`:

```
file = open('arquivo.txt')
try:
   conteudo = file.read()
finally:
   file.close()
```

Esse é um dos exemplos da PEP de 2005.

O bloco

`with`

A gramática

Inicialmente, vamos usar a forma sintática mais simples (antes da PEP 617 [2020]):

```
with_stmt ::=  "with" with_item ("," with_item)* ":" suite
with_item ::=  expression ["as" target]
```

Onde temos a tag `with` seguida por uma expressão e um `as` opcional.

```
with contexto:
    ...

with contexto() as var:
    # O resultado de `contexto` é referenciado em `var`
    ...
```

Alguns exemplos de uso [Estado global]

Além do já conhecido **open**, temos mais coisas na biblioteca padrão que usam gerenciamento de contexto e que podem ser divertidas de ver:

contextlib:

```
from contextlib import suppress

with suppress(Exception):
    1 / 0
    a = 1
    x.append(1)
```

Nenhuma exceção será levantada dentro desse bloco.

Note que aqui o estado global é alterado, não existe um "recurso"

Alguns exemplos de uso [Estado do módulo]

Além do já conhecido **open**, temos mais coisas na biblioteca padrão que usam gerenciamento de contexto e que podem ser divertidas de ver:

decimal:

```
from decimal import Decimal, localcontext

print(Decimal(1) / Decimal(7)) # 0.142857142857142857142857142857

with localcontext() as ctx:
    ctx.prec = 3
    print(Decimal(1) / Decimal(7)) # 0.143

print(Decimal(1) / Decimal(7)) # 0.142857142857142857142857142857
```

Somente no contexto, as operações de decimal terão 3 casas após o ponto. Ou seja, o estado do módulo `decimal` é alterado.

Alguns exemplos de uso [Estado de execução]

Além do já conhecido **open**, temos mais coisas na biblioteca padrão que usam gerenciamento de contexto e que podem ser divertidas de ver:

concurrent.futures:

```
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(max_workers=8) as executor:
    resultados = executor.map(lambda x: x * x, range(100))

print("Resultados:", list(resultados))
```

Aqui o sistema aloca 8 threads para resolver a operação e desaloca após o resultado. O estado de execução é alterado.

Alguns exemplos de uso [Estado de execução]

Além do já conhecido **open**, temos mais coisas na biblioteca padrão que usam gerenciamento de contexto e que podem ser divertidas de ver:

unittest.mock

```
from unittest.mock import patch

with patch('os.getcwd', return_value='/fake'):
    print(os.getcwd())
```

O contexto do módulo alvo do `patch` será alterado momentaneamente.

async `with`

Nem só de código síncrono vive o nosso código. Às vezes, é preciso garantir que algo seja gerenciado, mas que seja `async`.

```
async with context():
    ...

async with context() as c:
    ...
```

O bloco é exatamente igual, somente adicionando a palavra `async` antes.

Exemplo async

Um exemplo bastante básico:

```python
import asyncio

async def tarefa(n):
    await asyncio.sleep(n)
    print(f"Tarefa {n} concluída")

async def main():
    async with asyncio.timeout(1):
        await asyncio.gather(
            tarefa(0.5),
            tarefa(0.8),
        )

asyncio.run(main())
```

O bloco vai levantar uma exception caso o tempo passe de 1 segundo.

3.9+ o novo parser

Após a PEP 617, 2020, com a adição do novo parser. O bloco `with` pode também abrir mais de um contexto ao mesmo tempo.

Algo como:

```
with (
    open('input.txt') as input_file,
    open('output.txt', 'w') as output_file
):
    output_file.write(
        input_file.read()
    )
```

Permitindo que diversos gerenciadores compartilhem o mesmo contexto.

Esses arquivos juntos serão fechados ao final do bloco.

PEP 806 - Mixed sync/async context managers

Essa é uma proposta ainda em draft para 3.15

Embora o parser permita abrir vários contextos no mesmo **`with`**, misturar sync e async ainda não é possível.

```
async with acquire_lock() as lock:
    with temp_directory() as tmpdir:
        async with connect_to_db(cache=tmpdir) as db:
            with open('config.json', encoding='utf-8') as f:
```

Virando essa "massaroca".

PEP 806 - Mixed sync/async context managers

A proposta, ainda em draft, quer uma usabilidade mais fluida nesse caso:

```
with (
    async acquire_lock() as lock,
    temp_directory() as tmpdir,
    async connect_to_db(cache=tmpdir) as db,
    open('config.json', encoding='utf-8') as f,
):
```

Simplificando as operações conjuntas.

0

Protocolo

O protocolo

O "monta"/"desmonta" geraram o protocolo de "gerenciador de contexto" baseado em dois métodos. Os **`__enter__`** e **`__exit__`**.

Quando um objeto é usado como **`expressão`** do bloco with, ele precisa implementar o protocolo:

```
class Contexto:
    def __enter__(self):
        print('Entrando no contexto')
    def __exit__(self, *args):
        print('Saindo no contexto')
```

Qualquer objeto com esse protocolo já pode ser usado.

O uso

De forma simples:

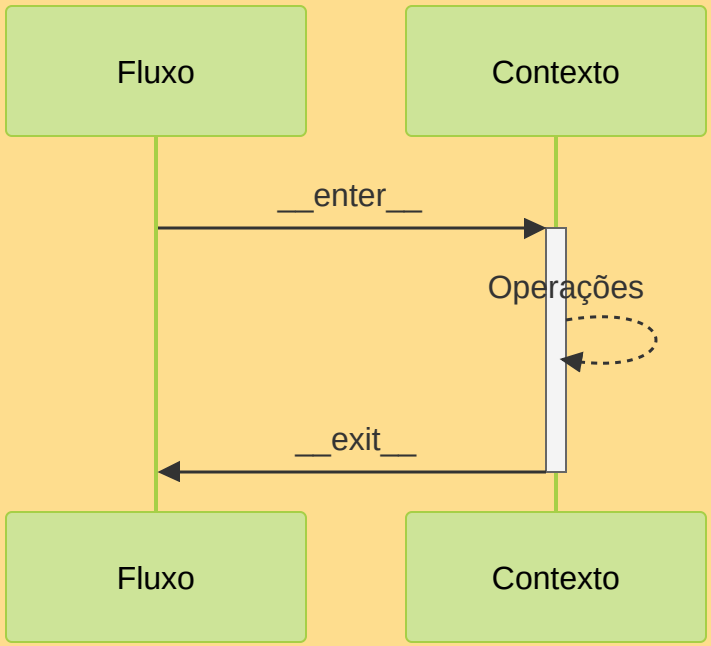
```
with Contexto():
    print('No contexto')
```

Isso vai resultar em:

```
$ python exemplo_01.py

Entrando no contexto
No contexto
Saindo no contexto
```

Por baixo

Todas as operações que ocorrem no bloco `with` estão sob a ação do contexto:



O que nos dá margem pra fazer coisas bem legais!

Um exemplo prático

Vamos supor que queremos "capturar" os `print`s:

```
import sys, io

class SuprimePrint:
    def __init__(self):
        self.out = io.StringIO()

    def __enter__(self):
        self.orig = sys.stdout
        sys.stdout = self.out
        return self

    def __exit__(self, *args):
        sys.stdout = self.orig
```

Usando

Algo simples como:

```
with SuprimePrint() as sp:
    print('teste!')
    print('teste2!')

print(f'Valores capturados: {sp.out.getvalue()}')
```

Nos resultaria em algo como:

```
$ python exemplo_00.py

Valores capturados: teste!
teste2!
```

Parâmetros para gerenciadores

Como `__enter__` não recebe parâmetros, eles devem ser passados ao `__init__` por exemplo:

```
class ManipulaArquivo:
    def __init__(self, target):
        self.target = target

    def __enter__(self):
        # retorna o arquivo para ser manipulado
        self.f = open(self.target)
        return self.f

    def __exit__(self, exc_type, exc_value, tb):
        self.f.close()
```

Simulando um **open** usando **open** xD

O `__exit__` e parâmetros

O `__exit__` recebe alguns parâmetros por padrão:

```python
def __exit__(self, exc_type, exc_value, tb):
    if exc_type:
        print("Erro:", exc_value)
    # Se True for retornado, a exception será suprimida no runtime
    return True
```

Só pra ficar evidente:

Parâmetro	O que contém	Exemplo
`exc_type`	A **classe** da exceção	`<class 'ZeroDivisionError'>`
`exc_value`	A **instância** (mensagem e dados) da exceção	`ZeroDivisionError('division by zero')`
`tb`	O **traceback** (pilha de chamadas)	Objeto `traceback`

Async!

O protocolo é basicamente igual para os gerenciadores de contexto assíncronos.

Adicionando o **a** no nome e tornando os métodos assíncronos:

```
class AContext:
    async def __aenter__(self):
        ...
    async def __aexit__(self, *args):
        ...
```

Podendo ser usado de forma simples:

```
async def main():
    async with AContext() as ac:
        ...
```

Context lib

E seus geradores :)

Contextlib

A Contextlib é uma das bibliotecas do python que facilita a criação de gerenciadores de contexto usando decoradores e geradores. Além de fornecer alguns gerenciadores prontos pra uso.

Decoradores base:

- `contextmanager`: Decora uma função geradora
- `asynccontextmanager`: Decora uma corrotina geradora

```python
from contextlib import contextmanager

@contextmanager
def context():
    ...  # enter
    yield
    ...  # exit
```

contextmanager

O nosso exemplo que suprime o `print`, pode ser escrito de forma mais simples:

```
import sys, io
from contextlib import contextmanager

@contextmanager
def suprime_print():
    out = io.StringIO()
    original_stdout = sys.stdout
    sys.stdout = out
    try:
        yield out
    finally:
        sys.stdout = original_stdout
```

A mesma coisa valeria para async, somente trocando o decorador.

Contextlib

Outros gerenciadores fornecidos pela contextlib:

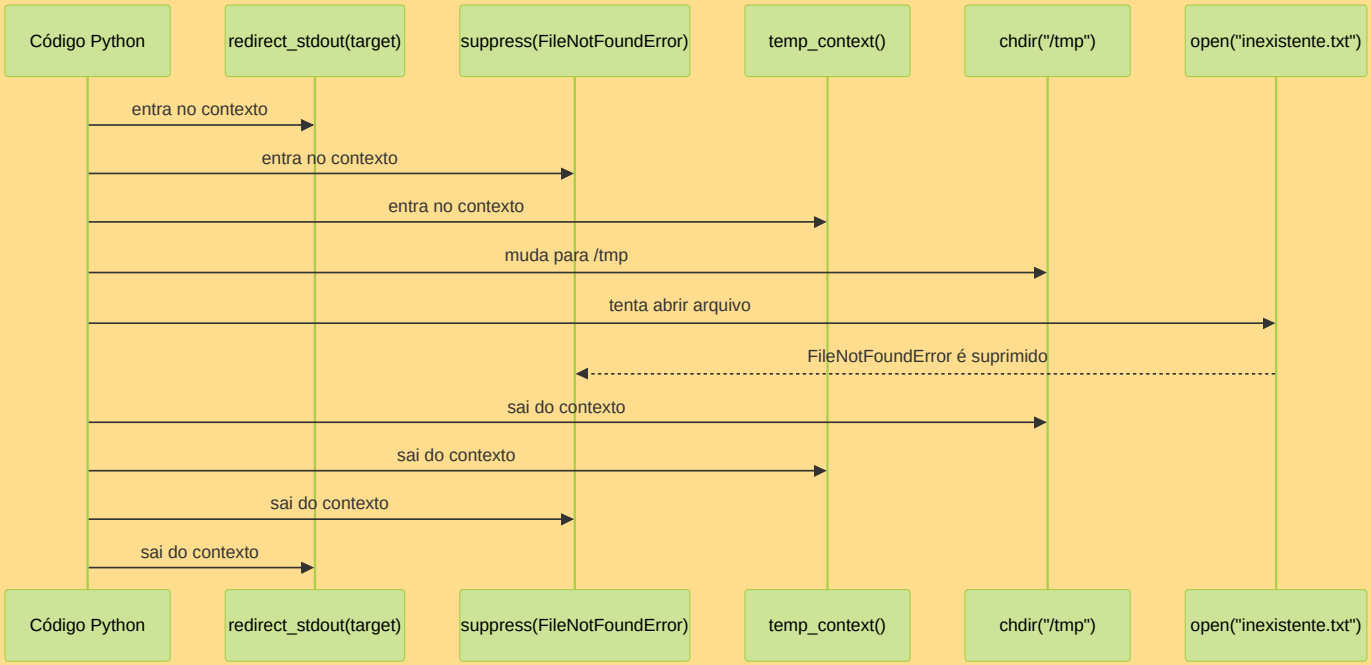
- `suppress(*exceptions)`:
  - Suprime exceções específicas dentro do bloco.
- `redirect_stdout(target)`:
  - Redireciona temporariamente o `sys.stdout`.
- `redirect_stderr(target)`:
  - Redireciona temporariamente o `sys.stderr`.
- `chdir(path)`:
  - Muda temporariamente o diretório de trabalho.
- `closing(obj)`:
  - Garante que o objeto seja fechado ao sair do bloco (`obj.close()`).

Tudo junto ao mesmo tempo xD

```
from contextlib import chdir, contextmanager, redirect_stdout, suppress

target = io.StringIO()

@contextmanager
def temp_context():
    print("Entrou")
    yield
    print("Saiu")  # Nunca vai sair no exemplo por conta do erro.

with (
    redirect_stdout(target), suppress(FileNotFoundError),
    temp_context(), chdir("/tmp")
):
    print("Diretório temporário ativo!")
    open("inexistente.txt")

print(target.getvalue())
```

Só pra ficar claro...

Ou nem tanto assim ...



Algumas coisas mais específicas

- `ExitStack()` / `AsyncExitStack()`:
  - Empilha múltiplos contextos dinamicamente.

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
```

Vai garantir que tudo que está em **`enter_context`** será finalizado. Uma forma de aninhamento

- `nullcontext([enter_result])`:
  - Gerenciador “vazio”, útil em código genérico.

```
def func(ignore_exceptions=False):
    if ignore_exceptions:
        cm = contextlib.suppress(Exception)
    else:
        cm = contextlib.nullcontext()
    with cm:
```

Referências

[1] “3. Data model - Asynchronous Context Managers”, Python documentation. Acesso em: 30 de outubro de 2025. [Online]. Disponível em: https://docs.python.org/3/reference/datamodel.html#asynchronous-context-managers

[2] “3. Data model - With Statement Context Managers”, Python documentation. Acesso em: 30 de outubro de 2025. [Online]. Disponível em: https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers

[3] “8. Compound statements”, Python documentation. Acesso em: 30 de outubro de 2025. [Online]. Disponível em: https://docs.python.org/3/reference/compound_stmts.html

[4] “10. Full Grammar specification”, Python documentation. Acesso em: 2 de novembro de 2025. [Online]. Disponível em: https://docs.python.org/3/reference/grammar.html

[5] “Built-in Types”, Python documentation. Acesso em: 30 de outubro de 2025. [Online]. Disponível em: https://docs.python.org/3/library/stdtypes.html

[6] “contextlib — Utilities for with-statement contexts”, Python documentation. Acesso em: 30 de outubro de 2025. [Online]. Disponível em: https://docs.python.org/3/library/contextlib.html

[7] “Contexto”, Michaelis On-Line. Acesso em: 31 de outubro de 2025. [Online]. Disponível em: https://michaelis.uol.com.br/moderno-portugues/busca/portugues-brasileiro/contexto/

[8] “PEP 343 – The ‘with’ Statement | peps.python.org”, Python Enhancement Proposals (PEPs). Acesso em: 31 de outubro de 2025. [Online]. Disponível em: https://peps.python.org/pep-0343/

[9] “PEP 617 – New PEG parser for CPython | peps.python.org”, Python Enhancement Proposals (PEPs). Acesso em: 2 de novembro de 2025. [Online]. Disponível em: https://peps.python.org/pep-0617/

[10] “PEP 806 – Mixed sync/async context managers with precise async marking | peps.python.org”, Python Enhancement Proposals (PEPs). Acesso em: 31 de outubro de 2025. [Online]. Disponível em: https://peps.python.org/pep-0806/

[11] “Defer - Learn Swift By Examples”. Acesso em: 1º de novembro de 2025. [Online]. Disponível em: https://learn-swift.com/defer/

[12] “Defer, Panic, and Recover - The Go Programming Language”. Acesso em: 1º de novembro de 2025. [Online]. Disponível em: https://go.dev/blog/defer-panic-and-recover

[13] “exceptions - Documentation for Ruby 3.5”. Acesso em: 1º de novembro de 2025. [Online]. Disponível em: https://docs.ruby-lang.org/en/master/syntax/exceptions_rdoc.html

[14] dotnet-bot, “IDisposable Interface (System)”. Acesso em: 1º de novembro de 2025. [Online]. Disponível em: https://learn.microsoft.com/en-us/dotnet/api/system.idisposable?view=net-9.0

[15] “The try-with-resources Statement (The Java™ Tutorials > Essential Java Classes > Exceptions)”. Acesso em: 1º de novembro de 2025. [Online]. Disponível em: https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html