Novidades do python 3.14

Live de Python # 299

Gostaria de agradecer ao time do i18n brasileiro. Enquanto montava essa live, uns 90% do que li para montar já estava traduzido.

Uma coisa importante

Gostaria de pedir aqui no chat um agradecimento coletivo <3

Uma coisa importante

Roreiro

1. Gerais

Melhorias, debugers, cores, ...

2. Plafaformas

Windows, Linux, Mac, ...

3. Bibliotecas

pathlib, uuid, zstandard, annotationlib

4. Sintaxe

Alterações e coisas novas

5. Intepretador

Coisas novas do cpython

6. O que não tem caixa

O que não coube em nenhum grupo xD

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto

Albano Maywitz, Alexandre Costa, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Alfredo Neto, Alynnefs, Alysson Oliveira, Andre Makoski, André Oliveira, Andre Paula, Antonio Filho, Apc 16, Apolo Ferreira, Artur_farias_, Aurelio Costa, Azmovi, Belisa Arnhold, Beltzery, Bernardo Fontes, Bernarducs, Biancarosa, Brisa Nascimento, Bruno Batista, Bruno Bereoff, Bruno Freitas, Bruno Ramos, Bruno Russian, Brunu, Canibasami, Caoptic, Carlos Gonçalves, Carlos Henrique, Cauã Oliveira, Celio Araujo, Christian Fischer, Claudemir Cruz, Cleiton Fonseca, Controlado, Curtos Treino, Daniel Aguiar, Daniel Brito, Daniel Bruno, Daniel Souza, Daniel Wojcickoski, Danilo Boas, Danilo Silva, Darcio Alberico, Darcioalberico_sp, David Couto, David Frazao, Dh44s, Diego Guimarães, Dilan Nery, Dunossauro, Edgar, Elias Soares, Emerson Rafael, Érico Andrei, Esdras, Everton Silva, Ewertonbello, Fábio Belotto, Fabio Faria, Fabiokleis, Felipe Adeildo, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Fernandocelmer, Ferrabras, Fichele Marias, Fightorfall, Francisco Aclima, Franklin Sousa, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Giuliano Silva, Glauber Duma, Gleidson Costa, Gnomo Nimp, Grinaode, Guibeira, Guilherme Felitti, Guilherme Ostrock, Gustavo Pedrosa, Gustavo Suto, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Henrique Machado, Henriquesebastiao, Herian Cavalcante, Hiago Couto, Idlelive, Igor Souza, Ivan Santiago, Ivansantiagojr, Janael Pinheiro, Jean Victor, Jeferson Vitalino, Jefferson Antunes, Jerry Ubiratan, Jhonata Medeiros, João Pena, Joao Rocha, 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, Kakaroto, Killfacept, Knaka, Krisquee, Laraalvv, Leandro Pina, Leandro Vieira, Leonardo Mello, Leonardo Nazareth, Lilian Pires, Lisandro Pires, Lucas Carderelli, Lucas Castro, Lucasgcode, Lucas Mello, 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, Mate65br, Mateusamorim96, Matheus Mendez, Matheus Vian, Medalutadorespacialx, Michael Santos, Mlevi Lsantos, Murilo Carvalho, Nhambu, Oopaze, Otávio Carneiro, Patrick Felipe, Programming, Rafael Ferreira, Rafael Fontenelle, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Ramon Lobo, Renan, Renan Sebastião, Renato José, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Richard Sousa, Rinaldo Magalhaes, Robsonpiere, Rodrigo Barretos, Rodrigo Santana, Rodrigo Vieira, Rogério Nogueira, Rui Jr, Samael Picoli, Samanta Cicilia, Santhiago Cristiano, Scrimf00x, Scrimfx, Shinodinho, Shirakawa, Sommellier_sr, Tarcísio Miranda, Tenorio, Téo Calvo, Teomewhy, Thamires Betin, Tharles Andrade, 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, Will_sca, Xxxxxxxxx, Yannzin, Zero! Studio

Obrigado você <3

Melhorias e novidades

Gerais

Melhoria nas mensagens de erro

Desde a versão 3.10 o python vem recebendo melhorias em mensagens de erro.

A versão 3.14 foi a que recebeu mais novos casos em todas as versões:

- Sugestões de palavras chave
- Erros de unpack
- Strings dentro de strings
- Erros com `as`
- Erros com hashables
- Prefixos incompatíveis de strings
- `if` como expressão + palavras reservadas
- Palavras chave após `else`

Sugestões para palavras-chave

Agora as palavras-chave tiveram os erros melhorados.

3.14:

```
>>> forr x in [1, 2, 3]:
  File "<python-input-1>", line 1
    forr x in [1, 2, 3]:
    ^^^^
SyntaxError: invalid syntax. Did you mean 'for'?
```

3.13:

```
>>> forr x in [1, 2, 3]:
  File "<python-input-1>", line 1
    forr x in [1, 2, 3]:
         ^
SyntaxError: invalid syntax
```

Erros de unpack

Antes só mostrava quantos eram esperados, mas não quantos vieram:

```
>>> a, b = 1, 2, 3
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    a, b = 1, 2, 3
    ^^^^
ValueError: too many values to unpack (expected 2, got 3)
```

Erro anterior:

```
ValueError: too many values to unpack (expected 2)
```

Prefixos de strings

Agora, quando os prefixos são incompatíveis, o erro diz que eles não podem ser usados em conjunto

```
>>> bt''
  File "<python-input-3>", line 1
    bt''
    ^^
SyntaxError: 'b' and 't' prefixes are incompatible
```

`as` e hashables

O `as`:

```
import ast as arr[0]
  File "<python-input-1>", line 1
    import ast as arr[0]
                  ^^^^^^
SyntaxError: cannot use subscript as import target
```

Erros com hashables:

```
>>> {['chave']: 'valor'}
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    {['chave']: 'valor'}
TypeError: cannot use 'list' as a dict key (unhashable type: 'list')
```

`elif` após `else`

Agora temos erros para quando a ordem das instruções não é respeitada do bloco `if`:

```
>>> if val:
...     ...
... else:
...     ...
... elif not val:
...     ...
...
  File "<python-input-5>", line 5
    elif not val:
    ^^^^
SyntaxError: 'elif' block follows an 'else' block
```

expressões `if` e keywords

Agora, quando qualquer palavra chave for usada no lugar das expressões, uma mensagem de erro específica será retornada:

```
>>> 1 if True else pass
  File "<python-input-6>", line 1
    1 if True else pass
                   ^^^^
SyntaxError: expected expression after 'else', but statement is given


>>> pass if True else False
  File "<python-input-7>", line 1
    pass if True else False
    ^^^^
SyntaxError: expected expression before 'if', but statement is given
```

Debuggers, Debuggers

Debug
gers

Debugger remoto

SIM! YAY! Agora dá pra usar o pdb em **outro processo** enquanto ele está sendo executado.

```
# xpto.py
from time import sleep

d = {}

while True:
    print(f'loop {d=}')
    sleep(5)
```

```
# pegar o processo
ps aux | grep xpto.py
# Ativar o debugger no processo
python -m pdb -p {processo}
```

Debugger remoto

Mas, calma, não vai funcionar de cara xD

Precisamos falar sobre permissões

Vamos tentar de novo:

```
# Ativar o debugger no processo
python -m pdb -p {processo}
```

Introspecção no asyncio

Agora é possível inspecionar, ao nível de processo, as tarefas que estão acontecendo em tempo de execução.

```
# pega o pid
ps aux | grep xpto.py

# olha a arvore
python -m asyncio pstree {processo}
```

Agora precisamos de um código rodando :)

Lambrando que... permissões

Introspecção no asyncio

continuando...

```
# asyncio_debug_test.py
from asyncio import gather, run, sleep as asleep
from time import sleep

async def maybe():
    print('maybe')
    await asleep(10)

async def task(n):
    print(f'task start - {n}')
    if n % 2:
        await maybe()
    await asleep(10)
    print(f'task end - {n}')

async def main():
    while True:
        sleep(3)
        await gather(*[task(x) for x in range(10)])

run(main())
```

E então...

Nem cabe na tela. hahaha

```
python -m asyncio pstree 937475
└── (T) Task-1
    └──  main /home/dunossauro/live_299/asyncio_debug_test.py:20
        ├── (T) Task-43
        │   └──  task /home/dunossauro/live_299/asyncio_debug_test.py:13
        │       └──  sleep /home/dunossauro/.../asyncio/tasks.py:702
        ├── (T) Task-45
        │   └──  task /home/dunossauro/live_299/asyncio_debug_test.py:13
        │       └──  sleep /home/dunossauro/.../asyncio/tasks.py:702
        ├── (T) Task-47
        │   └──  task /home/dunossauro/live_299/asyncio_debug_test.py:13
        │       └──  sleep /home/dunossauro/.../asyncio/tasks.py:702
        ├── (T) Task-49
        │   └──  task /home/dunossauro/live_299/asyncio_debug_test.py:13
        │       └──  sleep /home/dunossauro/.../asyncio/tasks.py:702
        └── (T) Task-51
            └──  task /home/dunossauro/live_299/asyncio_debug_test.py:13
                └──  sleep /home/dunossauro/.../asyncio/tasks.py:702
```

Outra forma de visualização

Outra forma de ver é o `ps`:

```
python -m asyncio ps {processo}
```

Que obviamente não vai saber na tela. Então, cortei muitos nomes, mas vamos ver no shell...

```
python -m asyncio ps 937475
tid     t id    task name   coroutine stack          a chain   a name    awaiter id
-------------------------------------------------------------------------------------
937475  0x7f74  Task-1      main                                          0x0
937475  0x7f74  Task-112    sleep -> task            main      Task-1    0x7f7432d24
937475  0x7f74  Task-113    sleep -> maybe -> task   main      Task-1    0x7f7432d24
937475  0x7f74  Task-114    sleep -> task            main      Task-1    0x7f7432d24
937475  0x7f74  Task-115    sleep -> maybe -> task   main      Task-1    0x7f7432d24
937475  0x7f74  Task-116    sleep -> task            main      Task-1    0x7f7432d24
```

REPL

Read
Eval
Print
Loop

Nada é mais
importante que isso...

CORES!

Cores!

Agora temos cores nos CLIs:

- Calendar
- Argparse
- json
- unittest

Calendar:

```
python -m calendar
```

Argparse:

```
python -m json.tool --help
```

Json

Também no json:

```
python -m json.tool test.json
{
    "key": "value",
    "pei": true,
    "au": 7
}
```

e no unittest:

```
python -m unittest teste_unittest.py
```

Plataforma

Onde roda
python 3.14?

Plataformas

PEP 11:

- Emscripten (PEP 776)
  - Falhas ainda não bloqueiam o build
  - Não tem distribuição oficial (use o pyodide)
- Android agora tem build oficial
  - Atualmente usado somente pelo briefcase (acho :)

Biblio tecas

pathlib, compression, uuid, annotationlib

pathlib

Agora a `pathlib` faz as operações do `shutil`, como:

Copia:

```
>>> Path('test.json').copy('test2.json')
Path('test2.json')
```

Move:

```
>>> Path('test.json').move('test2.json')
Path('test2.json')
```

Também entraram métodos como `move|copy_into()`. Que copia ou move o arquivo para um diretório existente específico.

UUID

Suporte para uuids 6, 7 e 8.

Podem ser usados via CLI:

```
python -m uuid -u uuid8
1f2eaa8f-a16b-82c9-8c4d-2d100676c79c
```

Ou então importando a biblioteca `uuid`:

```
>>> from uuid import uuid7
>>> uuid7()
UUID('0199844f-db9b-70de-8ea1-9ba2dc5b3f76')
```

Pacote `compression`

Agora existe um pacote chamado compression.

Que agrupa todos os módulos de compressão: `bz2`, `gzip`, `lzma` e `zlib`:

```
from compression import gzip
import shutil

with (
    open('test.json', 'rb') as f_in,
    gzip.open('test.json.gz', 'wb') as f_out
):
    shutil.copyfileobj(f_in, f_out)
```

Dá documentação: "(...) não serão removidos sem um ciclo de descontinuação. O uso de módulos em compression é incentivado sempre que possível."

ps: `zipfile` não está nesse módulo, pois não é compressão. Será?

A nova `compression.zstd`

O módulo `compression` traz o `compression.zstd`. Um módulo completamente novo de compressão para `Zstandard`.

Um exemplo básico:

```
from compression import zstd

data = b'Um monte de dados...'
with zstd.open("file.zst", "w") as f:
    f.write(data)
```

Acredito que valha uma live no futuro somente sobre o módulo `compression`, já que são uma coisa unificada :)

Sério... abre uma issue...

A nova `annotationlib` - PEP 749

A `annotationlib` é uma biblioteca para fazer introspecção de tipos de forma mais facilitada:

```
from annotationlib import get_annotations

def xpto(a: 'int') -> 'float': ...

>>> get_annotations(xpto)
{'a': 'int', 'return': 'float'}

>>> get_annotations(xpto, eval_str=True)
{'a': <class 'int'>, 'return': <class 'float'>}
```

Mas... Vamos conversar sobre ela um pouco mais à frente, em `Sintaxe`.

Sintaxe

E suas novidades!

PEP 758

Agora, múltiplas exceptions não precisam mais dos `()` para definir um bloco `except`:

```
try:
    1/0
except ZeroDivisionError, ValueError:
    print('Deu ruim')
```

Isso vale para grupos de exceptions também:

```
try:
    async with asyncio.TaskGroup() as group:
        group.create_task(coro())
        group.create_task(coro())
except* ZeroDivisionError, ValueError:
    print('Deu ruim!')
```

PEP 765 – Sintax Warning no bloco `finally`

Agora, para evitar que as exceptions passem despercebidas quando o bloco `finally` suprimir um erro. Por exemplo, como um `return`:

```
def xpto():
    try:
        ...
        raise Exception('Deu ruim!')
    except Exception:
        raise KeyboardInterrupt('Deu ruim!')
    finally:  # O que vai rolar???
       return 'OK'
```

A resposta deve ser parecida com essa:

```
<python-input-3>:8: SyntaxWarning: 'return' in a 'finally' block
```

PEP 765 – Sintax Warning no bloco `finally`

Esse comportamento se manifesta com qualquer coisa que faça um "goto". Como:

- `return`
- `continue`
- `break`

```
>>> for x in range(3):
...     try:
...         1/0
...     finally:
...         print('é o que?')
...         continue
<python-input-23>:6: SyntaxWarning: 'continue' in a 'finally' block
é o que?
é o que?
é o que?
```

PEP 750 – Template strings

Vai ganhar uma live específica. Por agora, somente um exemplo:

```
from string.templatelib import Template, Interpolation

def html(template: Template) -> str:
    parts = ''
    for item in template:
        if isinstance(item, str):
            parts += item
        elif isinstance(item, Interpolation):
            val = item.value
            if isinstance(val, dict):
                parts += ' '.join(f'{k}="{v}"' for k, v in val.items())

    return parts
```

Essa função processa um template e retorna uma string

PEP 750 – Template strings

Exemplo de uso:

```
attributes = {"src": "shrubbery.jpg", "alt": "looks nice"}
template = t"<img {attributes} />"

>>> html(template)
'<img src="shrubbery.jpg" alt="looks nice" />'
```

Como podemos ver, o objetivo das `t-strings` é que elas sejam processadas por uma função.

Tendo um comportamento diferente das `f-strings`, `u-strings` e `r-strings`.

PEP 749/649: Avaliando tipos tardiamente

Agora vai ser possível o que todos estavam esperando há anos:

```
class C:
    def xpto(self) -> D: ...

class D: ...
```

Os tipos agora não são mais calculados de forma ansiosa. Mas sim, *lazy*. O que quer dizer que o `obj.__annotations__` é um descritor, que será calculado quando for invocado.

PEP 749/649: Avaliando tipos tardiamente

Algo como:

```
class function:
    def __annotate__(format, /) -> dict:
        ... # implementar

    @property
    def __annotations__(self):
        if getattr(self, '__annotate__', None):
            return self.__annotate__(1)
        else:
            return {}
```

Ganhamos um novo dunder. O `__annotate__`, que é quem de fato "calcula" as anotações de tipos do objeto.

PEP 749/649: Avaliando tipos tardiamente

Como as anotações são feitas pelo `__annotate__`, precisamos entender os formatadores.

Os formatadores são definidos por um `Enum`:

- `VALUE = 1`: O valor real
- `VALUE_WITH_FAKE_GLOBALS = 2`: Uso interno do objeto
- `FORWARDREF = 3`: Uma referência de proxy para o objeto definido em outro lugar
- `STRING = 4`: O valor em string

```
>>> class C:
...     def xpto(self) -> D: ...
>>> annotationlib.get_annotations(
    C.xpto, format=annotationlib.Format.FORWARDREF
)
{'return': ForwardRef('D', owner=<function C.xpto at 0x7f7fef455fe0>)}
```

Novidades no

Inter pretador

Um novo tipo de interpretador

RECURSO AINDA EXPERIMENTAL

Nessa release ganhamos uma nova versão do interpretador: issue.

Como uma alternativa aos interpretadores já existentes, como o `goto`, o `switch-case` e o `uop`. Dessa vez usando `tail-call`.

Essa nova versão é opcional para compiladores mais modernos (clang 19+ e em x86-64 e AArch64), compilando com a flag `--with-tail-call-interp`.

Em uma média geométrica de 3 a 5% mais rápido no `pyperformance`.

O GCC ainda não oferece suporte a tail-call

PEP 734: múltiplos interpretadores

Agora, oficialmente, os múltiplos interpretadores têm uma versão estável da API.

Podem ser chamados diretamente via `concurrent`:

```
>>> from concurrent import interpreters
>>> i = interpreters.create()
>>> i.exec('print("pei!")')
pei!
```

Atualmente, todas as chamadas ainda precisam ser feitas via strings.

Os múltiplos interpretadores (subinterpretadores), por serem chamadas novas em memória, não têm o bloqueio da GIL.

Já fizemos uma live de python sobre esse assunto com o JS Bueno: https://youtu.be/PaTwb2ytFUg

PEP 734: múltiplos interpretadores

O módulo `concurrent.futures` agora também prove um novo `Pool` de execução de tarefas:

```
from concurrent.futures import InterpreterPoolExecutor as IPE

with IPE(max_workers=10) as pool:
    start = time.perf_counter()
    results = [pool.submit(task) for _ in range(10)]
    print([x.result() for x in results])
```

Onde `workers` são subinterpretadores e cada um resolve sua task isoladamente do interpretador principal.

O custo de memória é bastante grande, veremos isso no final dessa live.

PEP 744: Versões binárias para o just-in-time

Os binários oficiais para macOS e Windows agora incluem um compilador JIT (just-in-time) experimental. Embora não seja recomendado para uso em produção, ele pode ser testado definindo com a variável de ambiente `PYTHON_JIT=1`.

Para testar, só precisamos definir a variável:

```
PYTHON_JIT=1 python arquivo.py
```

O status atual do JIT pode ser visto nessa postagem do Ken

PEP 779: Python com threads livres é oficial

A PEP 703 introduziu o interpretador sem GIL como experimental na versão `3.13`, fase I.

A fase II foi definida na PEP 779, especificações para que o `no GIL`

Foram especificados os passos para o suporte oficial.

- Desempenho aceitável: No máximo 10-15% mais lento
- Uso aceitável de memória: 20%+ de consumo de memória
- APIs "comprovadamente" estáveis: para que as bibliotecas possam usar
- Documentação interna: Para que os core devs possam trabalhar

Todos esses critérios foram atendidos na versão `3.14`!

PS*: A fase III ainda terá os critérios definidos no futuro.

Para ver o status das bibliotecas que suportam o python sem a GIL clique aqui!

Uma pequena comparação

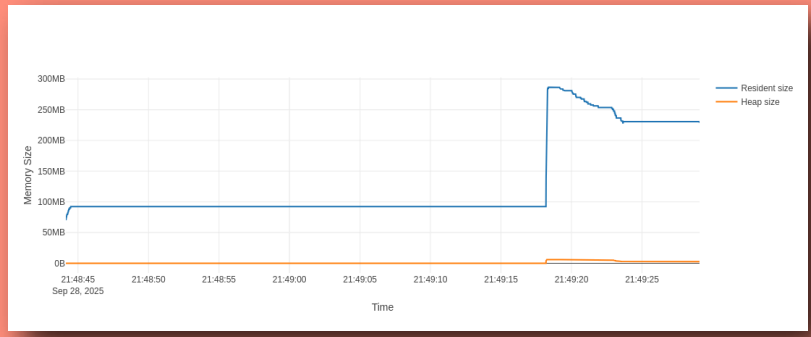
Só pra gente ver uma comparação mínima em ação:

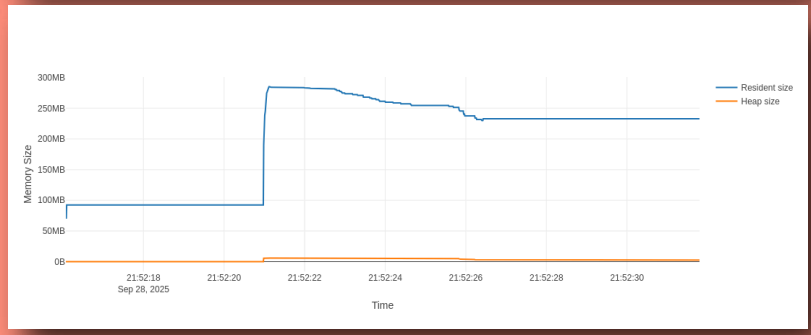
```
from concurrent.futures import ThreadPoolExecutor as TPE


with TPE(max_workers=10) as pool:
    start = time.perf_counter()
    results = [pool.submit(task) for _ in range(50)]


print([x.result() for x in results][-1])
print(f'Tempo passado (TPE): {time.perf_counter() - start}')
```

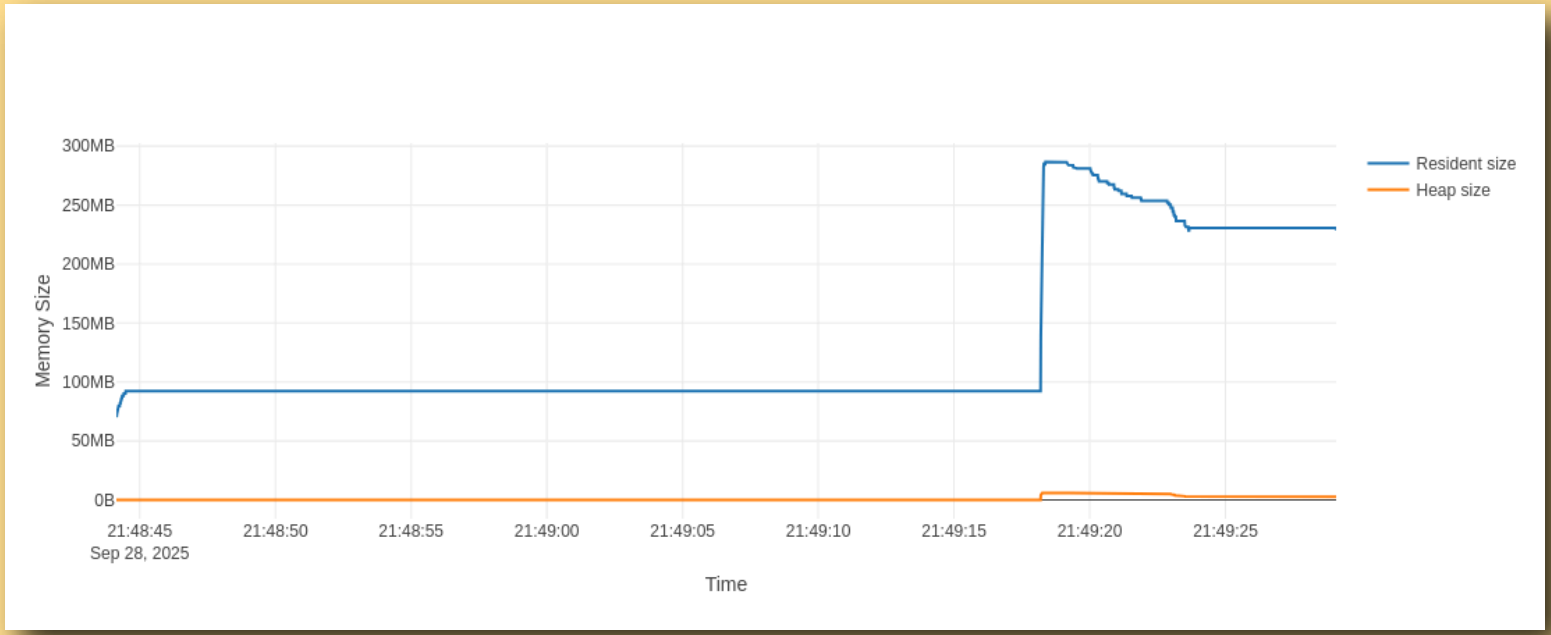
Versão	Tempo passado (TPE)
com gil	34.03950909300056
sem gil	4.888323546999345

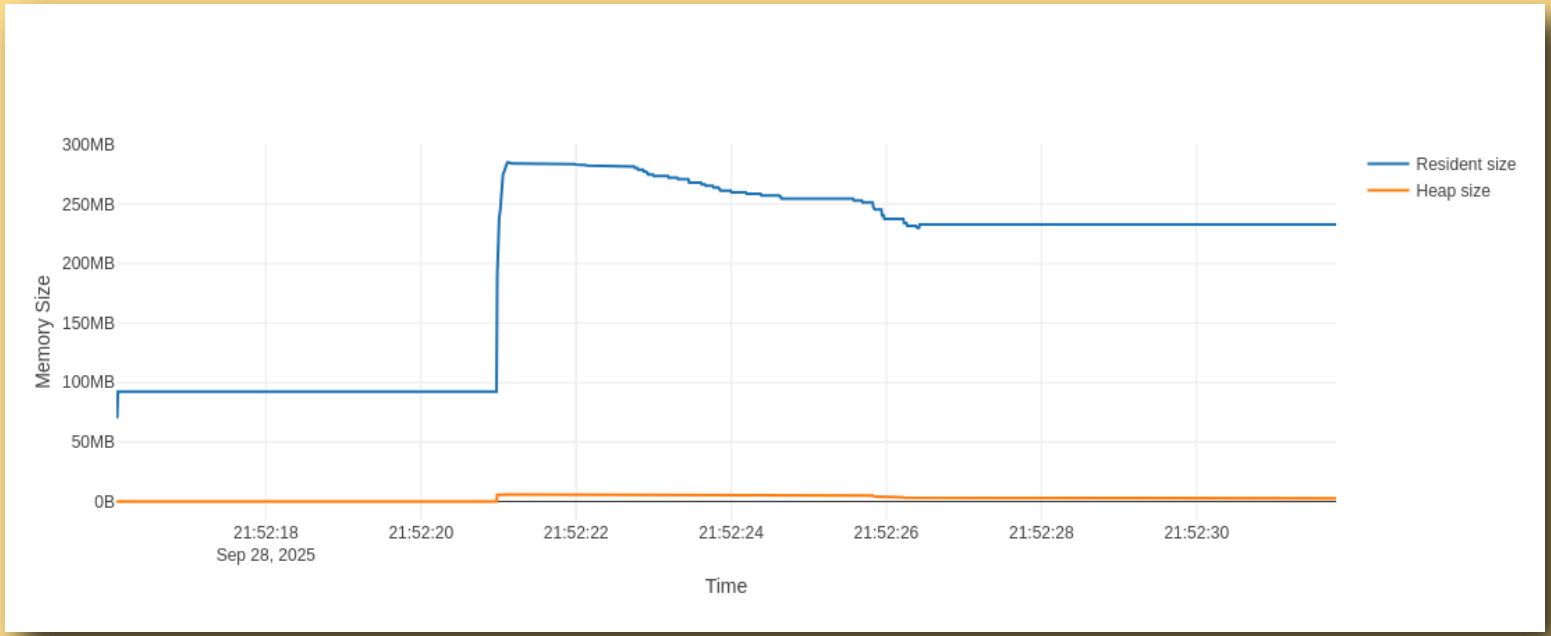
Comparando tempo e memória entre os Pools de futures

```python
import time
from datetime import datetime
from concurrent.futures import ThreadPoolExecutor as TPE
from concurrent.futures import ProcessPoolExecutor as PPE
from concurrent.futures import InterpreterPoolExecutor as IPE

MAX_WORKERS = 10
N_TASKS = 50

def task():
    n = 1
    for x in range(10_000_000):
        n += x + x * 2
    return n


def tpe():
    """ThreadPoolExecutor."""
    with TPE(max_workers=MAX_WORKERS) as pool:
        start = time.perf_counter()
        results = [pool.submit(task) for _ in range(N_TASKS)]

    print([x.result() for x in results][-1])
    print(f'Tempo passado (TPE): {time.perf_counter() - start}')


def ipe():
    """InterpreterPoolExecutor."""
    with IPE(max_workers=MAX_WORKERS) as pool:
        start = time.perf_counter()
        results = [pool.submit(task) for _ in range(N_TASKS)]

    print([x.result() for x in results][-1])
    print(f'Tempo passado (IPE): {time.perf_counter() - start}')


def ppe():
    """ProcessPoolExecutor."""
    with PPE(max_workers=MAX_WORKERS) as pool:
        start = time.perf_counter()
        results = [pool.submit(task) for _ in range(N_TASKS)]

    print([x.result() for x in results][-1])
    print(f'Tempo passado (PPE): {time.perf_counter() - start}')
```






300MB
250MB
200MB
150MB
100MB
50MB
0B
Memory Size
21:48:45
Sep 28, 2025
21:48:50
21:48:55
21:49:00
21:49:05
21:49:10
21:49:15
21:49:20
21:49:25
Time
Resident size
Heap size


300MB
250MB
200MB
150MB
100MB
50MB
0B
Memory Size
21:52:18
Sep 28, 2025
21:52:20
21:52:22
21:52:24
21:52:26
21:52:28
21:52:30
Time
Resident size
Heap size

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto