



Técnicas de Programação

Métodos para melhorar o desempenho do código

Métodos para melhorar o desempenho em Python

- Escolha do algoritmo
- Escolha da estrutura de dados
- Usar funções nativas (built-in)
- Compilar
- Código assíncrono
- Computação paralela e computação distribuída

Perfilamento do código

- Profilers conseguem indicar qual parte de uma função leva mais tempo para executar e fornece níveis de detalhamento
- Permitem **analisar o consumo de memória, a quantidade de chamadas a cada função e a hierarquia das execuções**, ajudando na otimização do código
- **Tipos de Profilers**
 - **1. Baseados em tempo (cProfile)**
Medem **quanto tempo cada função demora para rodar**.
Útil para encontrar **funções lentas e gargalos de CPU**.
 - **2. Baseados em memória (memory_profiler, MemRay)**
Analisam o **uso de memória ao longo do tempo**.
Identificam **vazamentos de memória e consumo excessivo**.
 - **3. Visualização de perfilamento (SnakeViz, py-spy)**
Criam gráficos e relatórios visuais para interpretar os dados mais facilmente.

cProfile

- Built-in
- Visão geral dos locais de gargalo em um script extenso

```
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
      1   0.002    0.002    0.050    0.050  script.py:5(mode_using_counter)
      1   0.030    0.030    0.030    0.030  <array_function> (random.randint)
      1   0.015    0.015    0.015    0.015  <built-in method Counter>
```

Interpretação dos dados:

- `ncalls`: Quantas vezes a função foi chamada.
- `tottime`: Tempo gasto **dentro** da função (sem contar chamadas internas).
- `cumtime`: Tempo total gasto, incluindo chamadas internas.

Código para a análise

```
def mode_using_counter(n_integers):  
    random_integers = np.random.randint(1, 100_000, n_integers)  
    c = Counter(random_integers)  
    return c.most_common(1)[0][0]
```

Usa `numpy.random.randint()` para gerar um **array de números inteiros aleatórios**.

- Os números gerados estão no intervalo **de 1 a 99.999** (100_000 é o limite superior exclusivo).
- A quantidade de números gerados é definida pelo argumento `n_integers`.
- Usa `Counter()` (do módulo `collections`) para **contar quantas vezes cada número aparece** no array.
- Usa `c.most_common(1)` para obter **a lista dos números mais frequentes**, retornando apenas **o mais comum** e `[0][0]` pega **o número mais frequente** da estrutura retornada, ou seja a moda.

```
import numpy as np
from collections import Counter
import cProfile
import pstats
import io

def mode_using_counter(n_integers):
    random_integers = np.random.randint(1,100_000,n_integers)
    c = Counter(random_integers)
    return c.most_common(1)[0][0]
    # Criando uma função para exibir o perfilamento
def profile_function():
    pr = cProfile.Profile()
    pr.enable()
    mode_using_counter(10_000_000)
    pr.disable()
    # Criando um buffer para armazenar os resultados
    s = io.StringIO()
    ps = pstats.Stats(pr, stream=s).sort_stats(pstats.SortKey.TIME)
    ps.print_stats()
    # Exibindo os resultados
    print(s.getvalue())
# Executando a análise de perfilamento
profile_function()
```

```
import numpy as np
from collections import Counter
import cProfile

def mode_using_counter(n_integers):
    random_integers = np.random.randint(1, 100_000, n_integers)
    c = Counter(random_integers)
    return c.most_common(1)[0][0]

cProfile.run('mode_using_counter(10_000_000)')
```

Visualizar graficamente com snakeviz

- Snakeviz: ferramenta que permite visualizar o perfilamento em um navegador
- Flask: framework web para rodar um servidor interno do snakeviz
- Pyngrok: biblioteca que cria um túnel para expor o servidor do snakeviz na internet

Intalação das dependências

```
!pip install snakeviz
```

```
!pip install flask
```

```
!pip install pyngrok
```

Importação das bibliotecas

```
from pyngrok import ngrok
import time
import numpy as np
from collections import Counter
import cProfile
```

pyngrok.ngrok → Permite criar um túnel para acessar o servidor do snakeviz fora do ambiente do Google Colab.

2. **time** → Usado para pausas no código (time.sleep()) e garantir que processos rodem corretamente.

3. **numpy** (np) → Biblioteca para manipulação de arrays e geração de números aleatórios.

4. **collections.Counter** → Ferramenta eficiente para contar elementos em listas e arrays.

5. **cProfile** → Módulo do Python usado para fazer o perfilamento de desempenho do código.

Código para a análise

```
def mode_using_counter(n_integers):  
    random_integers = np.random.randint(1, 100_000, n_integers)  
    c = Counter(random_integers)  
    return c.most_common(1)[0][0]
```

Usa `numpy.random.randint()` para gerar um **array de números inteiros aleatórios**.

- Os números gerados estão no intervalo **de 1 a 99.999** (100_000 é o limite superior exclusivo).
- A quantidade de números gerados é definida pelo argumento `n_integers`.
- Usa `Counter()` (do módulo `collections`) para **contar quantas vezes cada número aparece** no array.
- Usa `c.most_common(1)` para obter **a lista dos números mais frequentes**, retornando apenas **o mais comum** e `[0][0]` pega **o número mais frequente** da estrutura retornada, ou seja a moda.

Cria o perfilamento

```
profile_filename="profile_results.prof"
• cProfile.run("mode_using_counter(100000)", profile_filename)
```

Usa o módulo cProfile para **analisar o desempenho** da função mode_using_counter(100000).

- Salva os resultados no arquivo "profile_results.prof"

O primeiro argumento "mode_using_counter(100000)" **executa a função** como uma string.

- O segundo argumento profile_filename **define onde os resultados serão armazenados.**

Interpretação dos dados:

- ncalls: Quantas vezes a função foi chamada.
- tottime: Tempo gasto **dentro** da função (sem contar chamadas internas).
- cumtime: Tempo total gasto, incluindo chamadas internas.

Cria o servidor para o relatório

Versão colab

```
import subprocess

import time

print("Iniciando o servidor SnakeViz...")

server_process = subprocess.Popen(["snakeviz", "profile_results.prof", "--server", "--port", "8050"], stdout=subprocess.PIPE, stderr=subprocess.PIPE)

time.sleep(5)

print("Servidor SnakeViz rodando na porta 8050!")
```

subprocess.Popen() para **iniciar o servidor do SnakeViz** sem travar a célula do Colab.

- **snakeviz** → Inicie o SnakeViz.
- **profile_results.prof** → Use o arquivo de perfilamento salvo anteriormente.
- **--server** → Rode o SnakeViz como um servidor web.
- **--port 8050** → Configure o servidor para rodar na **porta 8050**.
- O Popen() permite rodar o processo **em segundo plano**, para que o restante do código continue executando normalmente.
- 🔍 **stdout=subprocess.PIPE, stderr=subprocess.PIPE**: Redireciona a **saída padrão (stdout)** e os **erros (stderr)** para que o código possa capturá-los se necessário.

Criando o acesso ao servidor

Versão colab

```
print("Iniciando autenticação Ngrok")
!ngrok authtoken 2uRly5lQc7Yj3jYjkAxid7DTJkG_3MbWcU4GzrYmREFmccp2J
print("Matando conexões antigas")
ngrok.kill()
try:
    print("Tentando abrir o túnel")
    public_url = ngrok.connect(8050)
    print("Túnel criado com sucesso")
    time.sleep(3)
    snakeviz_path = "/snakeviz/%2Fcontent%2Fprofile_results.prof"
    full_url = public_url.public_url + snakeviz_path
    print("Acesse SnakeViz em:", full_url)

except Exception as e:
    print("Erro ao iniciar Ngrok:", e)
```

Usando o Ngrok

Versão colab

Autentica o usuário no **Ngrok** usando o token fornecido.

- O token é necessário para permitir a criação de túneis públicos.
- Esse comando **só precisa ser executado uma vez** por sessão do Google Colab.
(<https://dashboard.ngrok.com/get-started/your-auth-token>)

- **Desconecta qualquer túnel Ngrok ativo** antes de iniciar um novo. Isso garante que não haja conflitos ao criar um novo túnel.

Tenta criar um túnel que redireciona a **porta 8050** (onde o SnakeViz está rodando) para um **endereço público na internet**

Cria um link **público** (exemplo: <https://1234abcd.ngrok.io>) que pode ser acessado de qualquer navegador

O snakeviz_path define **o caminho correto para visualizar os resultados do perfilamento** no servidor do SnakeViz.

Se houver **qualquer erro** ao criar o túnel, captura a exceção e exibe uma mensagem no console.

```
import numpy as np
import timeit
import time
from collections import Counter
import cProfile
import pstats

def mode_using_counter(list_of_numbers):
    c = Counter(list_of_numbers)
    return c.most_common(1)[0][0]
```



```
# Criando os números aleatórios
random_integers = np.random.randint(1, 1_000_000, 1_000_000)

# Criar o arquivo de perfil de desempenho
profiler = cProfile.Profile()
Profiler.enable()
mode_using_counter(random_integers)
profiler.disable()

# Salvar o perfil em um arquivo
profiler.dump_stats("profile_results.prof")
```

```
para rodar no terminal: snakeviz profile_results.prof
```

```
def slow_way_to_calculate_mode(list_of_numbers):  
    result_dict = {}  
    for i in list_of_numbers:  
        if i not in result_dict:  
            result_dict[i] = 1  
        else:  
            result_dict[i] += 1  
  
    mode_vals = []  
    max_frequency = max(result_dict.values())  
    for key, value in result_dict.items():  
        if value == max_frequency:  
            mode_vals.append(key)  
    return mode_vals
```

Implemente agora para
esta função e compare os
resultados

Usando Profiles em outra função intensiva

`nearest_prime(n_integers)`, que:

- Gera *n_integers* números aleatórios e para cada número, encontra o **primo mais próximo**, retornando o número mais frequente entre os primos encontrados

Rode a função seguindo e visualize e faça o perfilamento.

Encontrar o número primo mais próximo

is_prime(n) → Verifica se um número n é primo.

nearest_prime(n) → Retorna o **primo mais próximo** de n (para cima ou para baixo).

mode_nearest_prime(n_integers):

- Gera $n_integers$ números aleatórios.
- Encontra o primo mais próximo para cada número.
- Conta os primos mais frequentes e **retorna o mais comum**.

Com o perfilamento, verifique:

- Como **funções matematicamente intensivas** impactam o desempenho
- Como **loops aninhados** (busca de primos) influenciam no tempo de execução
- • Como **SnakeViz** pode mostrar **quais partes do código são mais pesadas**
- Quanto do tempo é gasto gerando a lista de números aleatórios e quando do tempo é alocado contando os números com Counter?

```
import numpy as np
from collections import Counter

def is_prime(n):
    """Verifica se um número é primo."""
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

```
def nearest_prime(n):  
    """Encontra o número primo mais próximo (para baixo ou para  
    cima)."""  
    if n < 2:  
        return 2 # O menor número primo  
    if is_prime(n):  
        return n # Se já for primo, retorna o próprio número  
  
    # Busca o primo mais próximo para cima e para baixo  
    lower, upper = n - 1, n + 1  
    while lower > 1 or upper < 2 * n:  
        if lower > 1 and is_prime(lower):  
            return lower  
        if is_prime(upper):  
            return upper  
        lower -= 1  
        upper += 1
```

```
def mode_nearest_prime(n_integers):  
    """Gera números aleatórios e retorna o primo mais frequente."""  
    random_integers = np.random.randint(1, 100_000, n_integers)  
  
    # Encontrar o primo mais próximo para cada número  
    nearest_primes = [nearest_prime(n) for n in random_integers]  
  
    # Contar a frequência dos primos encontrados  
    c = Counter(nearest_primes)  
  
    # Retorna o primo mais frequente  
    return c.most_common(1)[0][0]  
  
Profile_filename = "profile_results_prime.prof"  
cProfile.run("mode_nearest_prime(10000), profile_name)
```


Consumo de memória

```
import numpy as np
from collections import Counter
import psutil
import os

def mode_using_counter(n_integers):
    """Gera números aleatórios e retorna o mais frequente"""
    process = psutil.Process(os.getpid()) # Obter o processo atual
    mem_before = process.memory_info().rss / 1024 ** 2 # Memória antes (MB)
    random_integers = np.random.randint(1, 100_000, n_integers)
    c = Counter(random_integers)
    result = c.most_common(1)[0][0]
    mem_after = process.memory_info().rss / 1024 ** 2 # Memória depois (MB)

    print(f"🔍 Memória antes: {mem_before:.2f} MB")
    print(f"🔍 Memória depois: {mem_after:.2f} MB")
    print(f"📈 Diferença de memória: {mem_after - mem_before:.2f} MB")
    return result


Mode_using_counter(100_000)
```

Usa **psutil** para obter o consumo de memória do processo atual.

Mede a memória **antes e depois da execução da função**.

Exibe a diferença de memória consumida durante a execução.

Analizando o resultado



```
Uso de memória ao longo do tempo (MB): [117.5859375, 123.0, 123.0]  
np.int64(10390)
```

17.5859375 MB → **Memória antes de criar os números aleatórios.** Esse é o **uso inicial da memória** antes de alocar o array de números.

123.0 MB → **Memória após criar os números aleatórios.** Aumentou porque `np.random.randint()` criou um grande array de 100.000 números.

123.0 MB → **Memória após processar os números com Counter.** O uso de memória **não aumentou** após a contagem, indicando que `Counter()` usa pouca memória extra.

np.int64(10390) → **Resultado final da função.** Esse é o **número mais frequente** encontrado na lista gerada. O tipo `np.int64` significa que o número foi gerado usando NumPy, que usa tipos numéricos otimizados.

A criação de um grande array consome a maior parte da memória.

A contagem de frequência (`Counter`) não aumentou significativamente o consumo de memória.

O resultado final é apenas um número inteiro, sem impacto no consumo de memória.

Visualizando os dados

```
import matplotlib.pyplot as plt
memory_usage = [117.5859375, 123.0, 123.0]
steps = ["Antes do array", "Depois do array", "Depois  
do Counter"]
plt.plot(steps, memory_usage, marker="o",  
         linestyle="-")
plt.xlabel("Etapas da Execução")
plt.ylabel("Uso de Memória (MB)")
plt.title("Consumo de Memória Durante a Execução")
plt.show()
```